

Hadoop 2.X HDFS 源码剖析

徐鹏 著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书以 Hadoop 2.6.0 源码为基础,深入剖析了 HDFS 2.X 中各个模块的实现细节,包括 RPC 框架实现、Namenode 实现、Datanode 实现以及 HDFS 客户端实现等。本书一共有 5 章,其中第 1 章从总体上介绍了 HDFS 的组件、概念以及典型的流程,同时详细介绍了 HDFS 各个组件间 RPC 接口的定义。第 2 章介绍了 Hadoop RPC 框架的实现,Hadoop RPC 是 HDFS 各个组件间通信所依赖的底层框架,可以理解为 HDFS 的神经系统。第 3~5 章分别介绍了 Namenode、Datanode 以及 HDFS 客户端这三个组件的实现细节,同时穿插介绍了 HDFS 2.X 的新特性,例如 Namenode HA、Federation Namenode 等。

阅读本书可以帮助读者从架构设计与源码实现角度了解 HDFS 2.X,同时还能学习 HDFS 2.X 框架中优秀的设计思想、设计模式、Java 语言技巧以及编程规范等。这些对于读者全面提高自己的技术水平有很大的帮助。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Hadoop 2.X HDFS 源码剖析 / 徐鹏著. —北京: 电子工业出版社, 2016.3
ISBN 978-7-121-28155-6

I. ①H… II. ①徐… III. ①分布式文件系统—研究 IV. ①TP316

中国版本图书馆 CIP 数据核字(2016)第 027311 号

策划编辑: 张春雨

责任编辑: 葛 娜

印 刷: 三河市双峰印刷装订有限公司

装 订: 三河市双峰印刷装订有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×980 1/16 印张: 35.25 字数: 879 千字

版 次: 2016 年 3 月第 1 版

印 次: 2016 年 3 月第 1 次印刷

定 价: 108.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010) 88258888。

献给远老师，愿正年轻的你，永远保持一颗对世界好奇的心。

献给我的父母、妹妹，我无法用语言表达对你们的爱，以及你们对这个家庭默默付出的感恩。

前 言

今天 Hadoop 已经成为大数据处理中十分重要的平台，一个以 Hadoop 为基础的活跃的开源生态圈已经逐步形成，Hadoop 的应用也由最初的互联网行业发展到金融行业、电信行业、IT 设备商以及数以万计的中小企业。Hadoop 的 HDFS 组件（Hadoop 分布式文件系统）以及 MapReduce 组件分别为上层框架提供了分布式存储和计算的能力。

HDFS 作为 Hadoop 中解决分布式存储的基础组件，最早是根据 GFS (Google File System) 论文的概念模型来设计实现的。然而，随着 HDFS 上层框架的丰富以及应用场景的扩展，用户对 HDFS 的功能、性能、稳定性、扩展性以及可靠性的要求越来越高，HDFS 2.X 版本也就应运而生。相对于 HDFS 1.X, HDFS 2.X 版本提出了很多振奋人心的新特性，如 Namenode HA、Federation Namenode、集中式缓存、快照等。但令人惋惜的是，至今还没有一本能够深入剖析 HDFS 2.X 内部实现细节，以及介绍 HDFS 2.X 新特性的书籍。本书的出现填补了上述空白，它是国内第一本深入剖析 HDFS 2.X 源码实现的书籍。

本书以 Hadoop 2.6.0 源码为基础，深入剖析了 HDFS 2.X 中各个模块的实现细节，包括 RPC 框架实现、Namenode 实现、Datanode 实现以及 HDFS 客户端实现等。阅读本书可以帮助读者从架构设计与源码实现角度了解 HDFS 2.X，同时还能学习 HDFS 2.X 框架中优秀的设计思想、设计模式、Java 语言技巧以及编程规范等。这些对于读者全面提高自己的技术水平有很大的帮助。

如何阅读本书

由于篇幅原因，本书并没有介绍 HDFS 实现中的一些基础知识，例如 Java NIO、动态代理、protobuf 等。而是直接切入源码分析 HDFS 的设计与实现，同时介绍了一些经典的设计模式、Java 语言技巧在 HDFS 实现中的应用。希望读者在阅读本书之前，先搭建好源码环境，并了解相应的基础知识，这样学习效果会更好。

本书一共有 5 章，相互之间的联系比较紧密，有联系的小节都有注释标注，读者可以根据注释跳跃阅读。

第 1 章是 HDFS 概述，从总体上介绍了 HDFS 的组件、概念以及典型的流程，同时详细介绍了 HDFS 各个组件间 RPC 接口的定义。由于 HDFS 流程大都比较复杂，往往涉及多个组件的配合，读者在阅读后续的具体章节时，可以查阅本章内容获取某个流程的总体描述，以

及 RPC 接口的具体定义。

第 2 章介绍了 Hadoop RPC 框架的实现，Hadoop RPC 是 HDFS 各个组件间通信所依赖的底层框架，可以理解为 HDFS 的神经系统。通过阅读本章，读者可以学习到一个典型的分布式 RPC 框架的实现细节，在本章中会介绍较多的设计模式，编程模型以及语言技巧，请读者注重积累。

第 3~5 章分别介绍了 Namenode、Datanode 以及 HDFS 客户端这三个组件的实现细节，同时穿插介绍了 HDFS 2.X 的新特性，例如 Namenode HA 就放在 Namenode 章介绍，而 Federation Namenode 新特性对 Datanode 的修改比较多，所以就放在 Datanode 章介绍。

读者在阅读本书的过程中，如果发现有任何不当之处，烦请您将意见和建议发往邮箱 xupeng.bupt@gmail.com，不胜感激。

本书代码

本书分析的代码版本为 Hadoop 2.6.0，书中部分较长的代码做了省略，完整的代码请从官网 <http://hadoop.apache.org> 下载。

致谢

感谢互联网，感谢开源软件，感谢 Hadoop 社区，感谢本书引用文献的所有原作者，是你们为 Hadoop 爱好者打开了一扇大门。

感谢电子工业出版社博文视点的张春雨老师，是您的信任使得这本书的出版成为可能。同时还要感谢许多我不知道名字的编辑为本书最终出版所做的付出和努力。

感谢丁雷在百忙之中抽出时间对本书提出许多建设性意见，同时感谢左谱军、张德阳、闫飞翔以及张涛对本书的审阅。

2015 年写这本书时正是自己很困难的一段时期，我很感恩有许多朋友在生活、工作上给我帮助以及包容。感谢吴佳宁、刘文博，你们一直是第一个伸出援手的哥们。感谢郑晓彤、袁玮、贺子昂、李强、和紫东、刘丹、何一舟为我引荐机会，谢谢你们。

特别感谢远见，书的撰写过程是如此漫长，是你在这段时间里把自信、阳光和快乐传播给我，让我更加积极、勇敢和有信心。没有你，这本书永远无法完成。

最后感谢我的父母和妹妹，谢谢你们默默为我做出的牺牲和付出，你们永远是我前进的动力。

目 录

第 1 章 HDFS	1
1.1 HDFS 概述	1
1.1.1 HDFS 体系结构	1
1.1.2 HDFS 基本概念	2
1.2 HDFS 通信协议	4
1.2.1 Hadoop RPC 接口	4
1.2.2 流式接口	20
1.3 HDFS 主要流程	22
1.3.1 HDFS 客户端读流程	22
1.3.2 HDFS 客户端写流程	24
1.3.3 HDFS 客户端追加写流程	25
1.3.4 Datanode 启动、心跳以及执行名字节点指令流程	26
1.3.5 HA 切换流程	27
第 2 章 Hadoop RPC	29
2.1 概述	29
2.1.1 RPC 框架概述	29
2.1.2 Hadoop RPC 框架概述	30
2.2 Hadoop RPC 的使用	36
2.2.1 Hadoop RPC 使用概述	36
2.2.2 定义 RPC 协议	40
2.2.3 客户端获取 Proxy 对象	45
2.2.4 服务器获取 Server 对象	54
2.3 Hadoop RPC 实现	63
2.3.1 RPC 类实现	63
2.3.2 Client 类实现	64
2.3.3 Server 类实现	76

第 3 章	Namenode (名字节点)	88
3.1	文件系统目录树	88
3.1.1	INode 相关类	89
3.1.2	Feature 相关类	102
3.1.3	FSEditLog 类	117
3.1.4	FSImage 类	138
3.1.5	FSDirectory 类	158
3.2	数据块管理	162
3.2.1	Block、Replica、BlocksMap	162
3.2.2	数据块副本状态	167
3.2.3	BlockManager 类 (done)	177
3.3	数据节点管理	211
3.3.1	DatanodeDescriptor	212
3.3.2	DatanodeStorageInfo	214
3.3.3	DatanodeManager	217
3.4	租约管理	233
3.4.1	LeaseManager.Lease	233
3.4.2	LeaseManager	234
3.5	缓存管理	246
3.5.1	缓存概念	247
3.5.2	缓存管理命令	247
3.5.3	HDFS 集中式缓存架构	247
3.5.4	CacheManager 类实现	248
3.5.5	CacheReplicationMonitor	250
3.6	ClientProtocol 实现	251
3.6.1	创建文件	251
3.6.2	追加写文件	254
3.6.3	创建新的数据块	257
3.6.4	放弃数据块	265
3.6.5	关闭文件	266
3.7	Namenode 的启动和停止	268
3.7.1	安全模式	268
3.7.2	HDFS High Availability	276
3.7.3	名字节点的启动	301
3.7.4	名字节点的停止	306

第 4 章 Datanode (数据节点)	307
4.1 Datanode 逻辑结构	307
4.1.1 HDFS 1.X 架构	307
4.1.2 HDFS Federation	308
4.1.3 Datanode 逻辑结构	310
4.2 Datanode 存储	312
4.2.1 Datanode 升级机制	312
4.2.2 Datanode 磁盘存储结构	315
4.2.3 DataStorage 实现	317
4.3 文件系统数据集	334
4.3.1 Datanode 上数据块副本的状态	335
4.3.2 BlockPoolSlice 实现	335
4.3.3 FsVolumeImpl 实现	342
4.3.4 FsVolumeList 实现	345
4.3.5 FsDatasetImpl 实现	348
4.4 BlockPoolManager	375
4.4.1 BPServiceActor 实现	376
4.4.2 BPOfferService 实现	389
4.4.3 BlockPoolManager 实现	396
4.5 流式接口	398
4.5.1 DataTransferProtocol 定义	398
4.5.2 Sender 和 Receiver	399
4.5.3 DataXceiverServer	403
4.5.4 DataXceiver	406
4.5.5 读数据	408
4.5.6 写数据 (done)	423
4.5.7 数据块替换、数据块拷贝和读数据块校验	437
4.5.8 短路读操作	437
4.6 数据块扫描器	437
4.6.1 DataBlockScanner 实现	438
4.6.2 BlockPoolSliceScanner 实现	439
4.7 DirectoryScanner	442
4.8 DataNode 类的实现	443
4.8.1 DataNode 的启动	444
4.8.2 DataNode 的关闭	446

第 5 章	HDFS 客户端	447
5.1	DFSClient 实现	447
5.1.1	构造方法	448
5.1.2	关闭方法	449
5.1.3	文件系统管理与配置方法	450
5.1.4	HDFS 文件与目录操作方法	451
5.1.5	HDFS 文件读写方法	452
5.2	文件读操作与输入流	452
5.2.1	打开文件	452
5.2.2	读操作——DFSInputStream 实现	461
5.3	文件短路读操作	481
5.3.1	短路读共享内存	482
5.3.2	DataTransferProtocol	484
5.3.3	DFSClient 短路读操作流程	488
5.3.4	Datanode 短路读操作流程	509
5.4	文件写操作与输出流	512
5.4.1	创建文件	512
5.4.2	写操作——DFSOutputStream 实现	516
5.4.3	追加写操作	543
5.4.4	租约相关	546
5.4.5	关闭输出流	548
5.5	HDFS 常用工具	549
5.5.1	FsShell 实现	550
5.5.2	DFSAdmin 实现	552

第 1 章 HDFS

HDFS（Hadoop 分布式文件系统）是运行在通用硬件上的分布式文件系统。HDFS 提供了一个高度容错性和高吞吐量的海量数据存储解决方案。HDFS 已经在各种大型在线服务和大型存储系统中得到广泛应用，已经成为各大网站等在线服务公司的海量存储事实标准，多年来为网站客户提供了可靠、高效的服务。本章将对 HDFS 进行一个提纲挈领的介绍。

1.1 HDFS 概述

HDFS（Hadoop Distributed File System，Hadoop 分布式文件系统）最开始是作为 Apache Nutch 搜索引擎项目的基础架构而开发的，是 Apache Hadoop Core 项目的一部分。HDFS 被设计为可以运行在通用硬件（commodity hardware）上、提供流式数据操作、能够处理超大文件的分布式文件系统。HDFS 具有高度容错、高吞吐量、容易扩展、高可靠性等特征，为大型数据集的处理提供了一个强有力的工具。

1.1.1 HDFS 体系结构

HDFS 是一个主/从（Master/Slave）体系结构的分布式系统，如图 1-1 所示，HDFS 集群拥有一个 Namenode 和一些 Datanode，用户可以通过 HDFS 客户端同 Namenode 和 Datanodes 交互以访问文件系统。

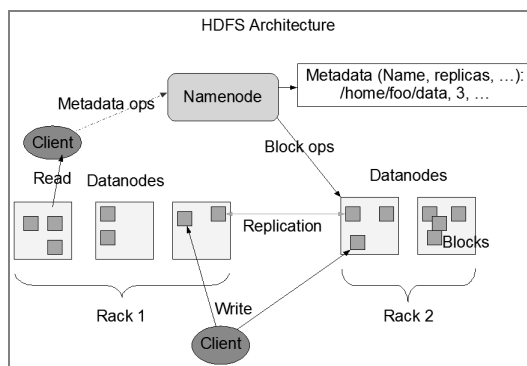


图 1-1 HDFS 体系结构示意图

在 HDFS 中，Namenode 是 HDFS 的 Master 节点，负责管理文件系统的命名空间(namespace)，以及数据块到具体 Datanode 节点的映射等信息。集群中的 Datanode 一般是一个节点一个，负责管理它所在节点上的存储。从内部看，一个文件其实被分成一个或多个数据块，这些块存储在一组 Datanode 上，Datanode 会以本地文件的形式保存这些数据块以及数据块的校验信息。

用户能够通过 HDFS 客户端发起读写 HDFS 文件的请求，同时还能通过 HDFS 客户端执行文件系统的命名空间操作，比如打开、关闭、重命名文件或目录。Namenode 会响应这些请求，更改命名空间以及数据块的映射信息，然后指导 Datanode 处理文件 HDFS 客户端的读写请求。

1.1.2 HDFS 基本概念

了解了 HDFS 整体架构后，我们再介绍 HDFS 中几个比较重要的概念。

1. 数据块 (Block)

HDFS 中数据块的概念与大部分 Linux 文件系统(ext2、ext3)的数据块概念相同，HDFS 文件是以数据块的形式存储的，数据块是 HDFS 文件处理的最小单元。由于 HDFS 文件往往比较大，同时为了最小化寻址开销，所以 HDFS 数据块也更大，默认是 128MB。HDFS 数据块会以文件的形式存储在数据节点的磁盘上。

在 HDFS 中，所有文件都会被切分成若干个数据块分布在数据节点上存储。同时由于 HDFS 会将同一个数据块冗余备份保存到不同的数据节点上（一个数据块默认保存 3 份），所以数据块的一个副本丢失了并不会影响这个数据块的访问。

在 HDFS 的读和写操作中，数据块都是最小单元。在读操作中，HDFS 客户端会首先到名字节点查找 HDFS 文件包含的数据块的位置信息，然后根据数据块的位置信息从数据节点读取数据。而在写操作中，HDFS 客户端也会首先从名字节点申请新的数据块，然后根据新申请数据块的位置信息建立数据流管道写数据。

2. 名字节点 (Namenode)

HDFS 是一个典型的 Master/Slave 结构的分布式系统，名字节点是 HDFS 主/从结构中的主节点。名字节点管理着文件系统的命名空间(namespace)，包括文件系统目录树、文件/目录信息以及文件的数据块索引，这些信息以两个文件的形式永久保存在名字节点的本地磁盘上，即命名空间镜像文件和编辑日志文件。同时名字节点还保存着数据块与数据节点的对应关系，这部分数据并不保存在名字节点的本地磁盘上，而是在名字节点启动时动态构建的。HDFS 客户端会通过名字节点获取上述信息，之后读写文件数据。

名字节点是 HDFS 中的单一故障点，如果名字节点丢失元数据或者损坏，文件系统将出现错误，甚至无法使用。为了解决名字节点的单点问题，Hadoop 2.X 版本引入了名字节点高可用性(HA)的支持。在 HA 实现中，同一个 HDFS 集群中会配置两个名字节点——活动名

字节点和备用名字节点。活动名字节点的内存元数据与备用名字节点是完全同步的，那么在活动名字节点发生故障而停止服务时，备用名字节点可以立即切换为活动状态，而不影响 HDFS 集群的服务。

名字节点的内存除了保存文件系统的命名空间外，还保存了文件系统中所有数据块与数据节点的对应关系，这意味着如果集群中文件数量过多时，名字节点的内存将成为限制系统横向扩展的瓶颈。为了解决这个问题，Hadoop 2.X 版本引入了联邦 HDFS 机制（HDFS Federation）。联邦 HDFS 机制允许添加名字节点以实现命名空间的扩展，其中每个名字节点都管理文件系统命名空间中的一部分，是一个独立的命名空间卷（namespace volume）。命名空间卷之间是相互独立的，两两之间并不相互通信，甚至其中一个名字节点失效了也不会影响由其他名字节点维护的命名空间的可用性。例如，一个名字节点可能管理/user 目录下的所有文件，而另一个名字节点可能管理/share 目录下的所有文件，这两个名字节点独立运行，互不影响。

HDFS 名字节点的相关内容我们会在第 3 章中介绍，包括 HDFS 元数据的管理，以及名字节点的 HA 机制等内容。而对于联邦 HDFS 机制，由于更改较多的是 Datanode 部分的代码，所以这部分内容我们会在第 4 章中介绍，请读者参考对应章节学习名字节点的实现。

3. 数据节点（Datanode）

数据节点是 HDFS 中的从节点，数据节点会根据 HDFS 客户端请求或者 Namenode 调度将新的数据块写入本地存储，或者读出本地存储上保存的数据块。

数据节点作为 HDFS 中的从节点，会不断地向名字节点发送心跳、数据块汇报以及缓存汇报，名字节点会通过心跳、数据块汇报以及缓存汇报的响应向数据节点发送指令，数据节点会执行这些指令，例如创建、删除或者复制数据等。

HDFS 数据节点的相关内容我们会在第 4 章中介绍。

4. 客户端

HDFS 提供了多种客户端接口供应用程序以及用户使用，包括命令行接口、浏览器接口以及代码 API 接口。用户通过这些接口可以很方便地使用 HDFS，而不需要考虑 HDFS 的实现细节。

而这些 HDFS 客户端接口的实现都是建立在 DFSClient 类的基础上的，DFSClient 类封装了客户端与 HDFS 其他节点间的复杂交互，我们会在第 5 章中介绍 HDFS 客户端的相关内容。

5. HDFS 通信协议

HDFS 作为一个分布式文件系统，它的某些流程是非常复杂的（例如读、写文件等典型流程），常常涉及数据节点、名字节点和客户端三者之间的配合、相互调用才能实现。为了降低节点间代码的耦合性，提高单个节点代码的内聚性，HDFS 将这些节点间的调用抽象成不同的接口。

HDFS 节点间的接口主要有两种类型。

- **Hadoop RPC 接口：**HDFS 中基于 Hadoop RPC 框架实现的接口。
- **流式接口：**HDFS 中基于 TCP 或者 HTTP 实现的接口。

由于 HDFS 通信协议部分内容比较多，我们会单独在本章的 HDFS 通信协议小节中介绍这两部分内容。

1.2 HDFS 通信协议

HDFS 通信协议抽象了 HDFS 各个节点之间的调用接口。由于本书后续章节只涉及各个节点的具体实现，所以这里将 HDFS 通信协议作为一个小节放在本章，做一个整体性的介绍。如果读者在后续章节中遇到通信协议相关的内容，可以查询本节的介绍。

1.2.1 Hadoop RPC 接口

Hadoop RPC 调用使得 HDFS 进程能够像本地调用一样调用另一个进程中的方法，并且可以传递 Java 基本类型或者自定义类作为参数，同时接收返回值。如果远程进程在调用过程中出现异常，本地进程也会收到对应的异常。目前 Hadoop RPC 调用是基于 Protobuf 实现的，我们会在第 2 章中介绍底层的具体实现，本节主要介绍 Hadoop RPC 接口的定义。Hadoop RPC 接口主要定义在 `org.apache.hadoop.hdfs.protocol` 包和 `org.apache.hadoop.hdfs.server.protocol` 包中，包括以下几个接口。

- **ClientProtocol：**ClientProtocol 定义了客户端与名字节点间的接口，这个接口定义的方法非常多，客户端对文件系统的所有操作都需要通过这个接口，同时客户端读、写文件等操作也需要先通过这个接口与 Namenode 协商之后，再进行数据块的读出和写入操作。
- **ClientDatanodeProtocol：**客户端与数据节点间的接口。ClientDatanodeProtocol 中定义的方法主要是用于客户端获取数据节点信息时调用，而真正的数据读写交互则是通过流式接口进行的。
- **DatanodeProtocol：**数据节点通过这个接口与名字节点通信，同时名字节点会通过这个接口中方法的返回值向数据节点下发指令。注意，这是名字节点与数据节点通信的唯一方式。这个接口非常重要，数据节点会通过这个接口向名字节点注册、汇报数据块的全量以及增量的存储情况。同时，名字节点也会通过这个接口中方法的返回值，将名字节点指令带回该数据块，根据这些指令，数据节点会执行数据块的复制、删除以及恢复操作。
- **InterDatanodeProtocol：**数据节点与数据节点间的接口，数据节点会通过这个接口和其他数据节点通信。这个接口主要用于数据块的恢复操作，以及同步数据节点上存储的数据块副本的信息。
- **NamenodeProtocol：**第二名字节点与名字节点间的接口。由于 Hadoop2.X 中引入了

HA 机制，检查点操作也不再由第二名字节点执行了，所以 NamenodeProtocol 我们就不详细介绍了。

- 其他接口：主要包括安全相关接口（RefreshAuthorizationPolicyProtocol、RefreshUserMappingsProtocol）、HA 相关接口（HAResourceProtocol）等。HA 相关接口的实现和定义我们将在第 3 章的 HA 小节中介绍。

下面我们重点介绍 ClientProtocol、ClientDatanodeProtocol、DatanodeProtocol、InterDatanodeProtocol 和 NamenodeProtocol 等接口的定义。

1. ClientProtocol

ClientProtocol 定义了所有由客户端发起的、由 Namenode 响应的操作。这个接口非常大，有 80 多个方法，我们把这个接口中的方法分为如下几类。

- HDFS 文件读相关的操作。
- HDFS 文件写以及追加写的相关操作。
- 管理 HDFS 命名空间（namespace）的相关操作。
- 系统问题与管理相关的操作。
- 快照相关的操作。
- 缓存相关的操作。
- 其他操作。

HDFS 文件读操作、HDFS 文件写与追加写操作，以及命名空间的管理操作，这三个部分都可以在 FileSystem 类中找到对应的方法，这些方法都是用来支持 Hadoop 文件系统实现的。对于系统问题与管理相关的操作，则是由 DFSAdmin 这个工具类发起的，其中的方法是用于支持管理员配置和管理 HDFS 的。而快照和缓存则都是 Hadoop2.X 中引入的新特性，ClientProtocol 中也有对应的方法用于支持这两个新特性。当然，ClientProtocol 中还包括安全、XAttr 等方法，这部分不是重点，我们就不再详细介绍了。

（1）读数据相关方法

ClientProtocol 中与客户端读取文件相关的方法主要有两个：getBlockLocations() 和 reportBadBlocks()。

客户端会调用 ClientProtocol.getBlockLocations() 方法获取 HDFS 文件指定范围内所有数据块的位置信息。这个方法参数是 HDFS 文件的文件名以及读取范围，返回值是文件指定范围内所有数据块的文件名以及它们的位置信息，使用 LocatedBlocks 对象封装。每个数据块的位置信息指的是存储这个数据块副本的所有 Datanode 的信息，这些 Datanode 会以与当前客户端的距离远近排序。客户端读取数据时，会首先调用 getBlockLocations() 方法获取 HDFS 文件的所有数据块的位置信息，然后客户端会根据这些位置信息从数据节点读取数据块。ClientProtocol.getBlockLocations() 方法的定义如下：

```
public LocatedBlocks getBlockLocations(String src,
                                       long offset,
                                       long length)
```

```
throws AccessControlException, FileNotFoundException,  
UnresolvedLinkException, IOException;
```

客户端会调用 `ClientProtocol.reportBadBlocks()` 方法向 `Namenode` 汇报错误的数据块。当客户端从数据节点读取数据块且发现数据块的校验和并不正确时，就会调用这个方法向 `Namenode` 汇报这个错误的数据块信息。`ClientProtocol.reportBadBlocks()` 方法的定义如下：

```
public void reportBadBlocks(LocatedBlock[] blocks) throws IOException;
```

(2) 写/追加写数据相关方法

在 HDFS 客户端操作中最重要的一部分就是写入一个新的 HDFS 文件，或者打开一个已有的 HDFS 文件并执行追加写操作。`ClientProtocol` 中定义了 8 个方法支持 HDFS 文件的写操作：`create()`、`append()`、`addBlock()`、`complete()`、`abandonBlock()`、`getAdditionalDatanodes()`、`updateBlockForPipeline()` 和 `updatePipeline()`。

`create()` 方法用于在 HDFS 的文件系统目录树中创建一个新的空文件，创建的路径由 `src` 参数指定。这个空文件创建后对于其他的客户端是“可读”的，但是这些客户端不能删除、重命名或者移动这个文件，直到这个文件被关闭或者租约过期。客户端写一个新的文件时，会首先调用 `create()` 方法在文件系统目录树中创建一个空文件，然后调用 `addBlock()` 方法获取存储文件数据的数据块的位置信息，最后客户端就可以根据位置信息建立数据流管道，向数据节点写入数据了。`create()` 方法的定义如下：

```
public HdfsFileStatus create(String src, FsPermission masked,  
    String clientName, EnumSetWritable<CreateFlag> flag,  
    boolean createParent, short replication, long blockSize,  
    CryptoProtocolVersion[] supportedVersions)  
    throws..., IOException;
```

`append()` 方法用于打开一个已有的文件，如果这个文件的最后一个数据块没有写满，则返回这个数据块的位置信息（使用 `LocatedBlock` 对象封装）；如果这个文件的最后一个数据块正好写满，则创建一个新的数据块并添加到这个文件中，然后返回这个新添加的数据块的位置信息。客户端追加写一个已有文件时，会先调用 `append()` 方法获取最后一个可写数据块的位置信息，然后建立数据流管道，并向数据节点写入追加的数据。如果客户端将这个数据块写满，与 `create()` 方法一样，客户端会调用 `addBlock()` 方法获取新的数据块。

```
public LocatedBlock append(String src, String clientName)  
    throws AccessControlException, DSQuotaExceededException,  
    FileNotFoundException, SafeModeException, UnresolvedLinkException,  
    SnapshotAccessControlException, IOException;
```

客户端调用 `addBlock()` 方法向指定文件添加一个新的数据块，并获取存储这个数据块副本的所有数据节点的位置信息（使用 `LocatedBlock` 对象封装）。要特别注意的是，调用 `addBlock()` 方法时还要传入上一个数据块的引用。`Namenode` 在分配新的数据块时，会顺便提交上一个数据块，这里 `previous` 参数就是上一个数据块的引用。`excludeNodes` 参数则是数据节点的黑名单，保存了客户端无法连接的一些数据节点，建议 `Namenode` 在分配保存数据块副本的数据节点时不要考虑这些节点。`favorNodes` 参数则是客户端所希望的保存数据块副本的数据节点。

点的列表。客户端调用 `addBlock()` 方法获取新的数据块的位置信息后，会建立到这些数据节点的数据流管道，并通过数据流管道将数据写入数据节点。`addBlock()` 方法的定义如下：

```
public LocatedBlock addBlock(String src, String clientName,
    ExtendedBlock previous, DatanodeInfo[] excludeNodes, long fileId,
    String[] favoredNodes)
    throws AccessControlException, FileNotFoundException,
    NotReplicatedYetException, SafeModeException, UnresolvedLinkException,
    IOException;
```

当客户端完成了整个文件的写入操作后，会调用 `complete()` 方法通知 Namenode。这个操作会提交新写入 HDFS 文件的所有数据块，当这些数据块的副本数量满足系统配置的最小副本系数（默认值为 1），也就是该文件的所有数据块至少有一个有效副本时，`complete()` 方法会返回 `true`，这时 Namenode 中文件的状态也会从构建中状态转换为正常状态；否则，`complete()` 会返回 `false`，客户端就需要重复调用 `complete()` 操作，直至该方法返回 `true`。

```
public boolean complete(String src, String clientName,
    ExtendedBlock last, long fileId)
    throws AccessControlException, FileNotFoundException, SafeModeException,
    UnresolvedLinkException, IOException;
```

上面描述的 5 个方法都是在正常流程时，客户端写文件必须调用的方法。但是对于一个分布式系统来说，写流程中涉及的任何节点都有可能出现故障。出现故障的情况也是需要考虑在内的，所以 `ClientProtocol` 定义了 `abandonBlock()`、`getAdditionalDatanode()`、`updateBlockForPipeline()` 以及 `updatePipeline()` 等方法，用于在异常情况下进行恢复操作。

客户端调用 `abandonBlock()` 方法放弃一个新申请的数据块。考虑下面这种情况：当客户端获取了一个新申请的数据块，发现无法建立到存储这个数据块副本的某些数据节点的连接时，会调用 `abandonBlock()` 方法通知名字节点放弃这个数据块，之后客户端会再次调用 `addBlock()` 方法获取新的数据块，并在传入参数时将无法连接的数据节点放入 `excludeNodes` 参数列表中，以避免 Namenode 将数据块的副本分配到该节点上，造成客户端再次无法连接这个节点的情况。

```
public void abandonBlock(ExtendedBlock b, long fileId,
    String src, String holder)
    throws AccessControlException, FileNotFoundException,
    UnresolvedLinkException, IOException;
```

`abandonBlock()` 方法用于处理客户端建立数据流管道时数据节点出现故障的情况。那么，如果客户端已经成功建立了数据流管道，在客户端写某个数据块时，存储这个数据块副本的某个数据节点出现了错误该如何处理呢？这个操作就比较复杂了，客户端首先会调用 `getAdditionalDatanode()` 方法向 Namenode 申请一个新的 Datanode 来替代出现故障的 Datanode。然后客户端会调用 `updateBlockForPipeline()` 方法向 Namenode 申请为这个数据块分配新的时间戳，这样故障节点上的没能写完整的数据块的时间戳就会过期，在后续的块汇报操作中会被删除。最后客户端就可以使用新的时间戳建立新的数据流管道，来执行对数据块的写操作了。数据流管道建立成功后，客户端还需要调用 `updatePipeline()` 方法更新 Namenode 中当前数据块的数据流管道信息。至此，一个完整的恢复操作结束。

上面我们描述的都是写在数据操作时数据节点发生故障的情况，包括了数据流管道建立时以及建立后数据节点发生故障的情况。在写数据的过程中，Client 节点也有可能在任意时刻发生故障，为了预防这种情况，对于任意一个 Client 打开的文件都需要 Client 定期调用 ClientProtocol.renewLease() 方法更新租约（关于租约请参考第 3 章中租约相关小节）。如果 Namenode 长时间没有收到 Client 的租约更新消息，就会认为 Client 发生故障，这时就会触发一次租约恢复操作，关闭文件并且同步所有数据节点上这个文件数据块的状态，确保 HDFS 系统中这个文件是正确且一致保存的。

如果在写操作时，名字节点发生故障该如何处理呢？这就要涉及 HDFS 的 HA 架构了，请读者参考第 3 章的 HA 部分。

（3）命名空间管理的相关方法

ClientProtocol 中有很重要的一部分操作是对 Namenode 命名空间的修改。我们知道 FileSystem 类也定义了对文件系统命名空间修改操作的 API（FileSystem 类抽象了一个文件系统对外提供的 API 接口），HDFS 则满足 FileSystem 类抽象的所有方法。表 1-1 总结了 FileSystem API 与 ClientProtocol 接口的对应关系。

表 1-1 FileSystem API 与 ClientProtocol 接口的对应关系

Hadoop FileSystem 操作	ClientProtocol 对应的接口	描 述
FileSystem.rename2()	rename2()	更改文件/目录名称
FileSystem.concat()	concat()	将两个已有文件拼接成一个
FileSystem.delete()	delete()	从文件系统中删除指定文件或者目录
FileSystem.mkdirs()	mkdirs()	以指定名称和权限在文件系统中创建目录
FileSystem.listStatus()	getListing()	读取一个指定目录下的所有项目
FileSystem.set* ()	setPermission() setOwner() setTimes() setReplication()	修改文件属性。分别用于修改文件权限、文件主/组、文件修改时间/访问时间以及文件的副本系数
FileSystem.listCorruptFileBlocks()	listCorruptFileBlocks()	获取文件系统中损坏文件的一部分，如果想要获取文件系统中所有损坏的文件，则循环调用这个方法
FileSystem.getFileStatus	getFileInfo()	获取文件/目录的属性
FileSystem.getFileLinkStatus	getFileLinkStatus()	获取文件/目录的属性，如果文件指向一个符号链接，则返回这个符号链接的信息
DistributedFileSystem.isFileClosed	isFileClosed()	判断指定文件是否关闭了
FileSystem.getContentSummary	getContentSummary()	获取文件/目录使用的存储空间信息
FileSystem.createSymlink()	createSymlink()	对于已经存在的文件创建符号链接
resolveLink()	getLinkTarget()	获取指定符号链接指向目标

通过表 1-1 我们可以看出，ClientProtocol 中涉及的命名空间管理的方法都有与之对应的 HDFS 文件系统 API，且方法的名称和参数很相近。在这里我们以 setReplication() 为例，

setReplication()方法在 ClientProtocol 中的定义是：

```
public boolean setReplication(String src, short replication)
    throws AccessControlException, DSQuotaExceededException,
        FileNotFoundException, SafeModeException, UnresolvedLinkException,
        SnapshotAccessControlException, IOException;
```

在 FileSystem 接口中的定义是：

```
public boolean setReplication(Path src, short replication)
    throws IOException {
    return true;
}
```

可以看到，setReplication()方法在 FileSystem 和 ClientProtocol 中定义的方法名及参数都很接近，大部分情况下 ClientProtocol 接口方法定义的参数更多，可以很好地支持 FileSystemAPI 定义的操作。

(4) 系统问题与管理操作

ClientProtocol 中另一个重要的部分就是支持 DFSAdmin 工具的接口方法，DFSAdmin 是供 HDFS 管理员管理 HDFS 集群的命令行工具。一个典型的 dfsadmin 命令如下所示，管理员可以添加不同的参数以触发 HDFS 进行相应的操作。

```
hdfs dfsadmin[参数]
```

表 1-2 给出了 ClientProtocol 中定义的接口方法与 dfsadmin 命令参数之间的对应关系，我们会重点讲解几个比较重要的方法。

表 1-2 ClientProtocol 中定义的接口方法与 dfsadmin 命令参数之间的对应关系

ClientProtocol 接口	dfsadmin 命令参数
getStatus()	用于获取文件系统状态信息，包括磁盘使用情况、复制数据块的数量、损坏数据块的数量、丢失数据块的数量等。对应于 dfsadmin 命令`-report`选项
getDatanodeReport()	获取集群中存活的、死亡的或者所有的数据节点信息。对应于 dfsadmin 命令`-report`选项
getDatanodeStorageReport()	获取数据节点上所有存储的信息
setSafeMode()	用于进入、离开安全模式，或者获取当前安全模式的状态。这个方法我们会在第 3 章的安全模式小节中详细介绍。对应于 dfsadmin 命令`-safemode`选项
saveNamespace()	将 Namenode 内存中的命名空间保存至新的 fsimage 中，并且重置 editlog。对应于 dfsadmin 命令`-saveNamespace`选项。注意，执行这个操作要求必须是处于安全模式中
rollEdits()	重置 editlog，也就是关闭当前正在写入的 editlog 文件，开启一个新的 editlog 文件。对应于 dfsadmin 命令`-rollEdits`选项。注意，执行这个操作要求必须是处于安全模式中
restoreFailedStorage()	用于当失败的（failed）存储变得可用时，设置是否对这个存储上保存的副本进行恢复操作。对应于 dfsadmin 命令`-restoreFailedStorage`选项
refreshNodes()	触发 Namenode 重新读取 include/exclude 文件。对应于 dfsadmin 命令`-refreshNodes`选项
finalizeUpgrade()	提交 Namenode 的升级操作。对应于 dfsadmin 命令`-finalizeUpgrade`选项
rollingUpgrade()	触发 Namenode 进行升级操作。对应于 dfsadmin 命令`-rollingUpgrade`选项

续表

ClientProtocol 接口	dfsadmin 命令参数
metaSave()	将 Namenode 中主要的数据结构保存到指定文件中，包括同 Namenode 心跳过的 Datanode、等待复制的数据块、等待删除的数据块、当前正在复制的数据块等信息。对应于 dfsadmin 命令`-metasave` 选项
setBalancerBandwidth()	更改 Datanode 在进行数据块平衡操作时所占用的带宽。调用这个命令设置的带宽值会覆盖 dfs.balance.bandwidthPerSec 配置项配置的带宽值。对应于 dfsadmin 命令`-setBalancerBandwidth` 选项
setQuota()	设置目录中的文件/目录的数量配额，以及文件大小的配额。对应于 dfsadmin 命令`-setQuota`、`-clrQuota`、`-setSpaceQuota`和`-clrSpaceQuota` 选项，这 4 个选项底层都是通过 setQuota()触发 Namenode 操作的

首先看一下 setSafeMode()方法，这里涉及一个非常重要的概念——安全模式。安全模式是 Namenode 的一种状态，处于安全模式中的 Namenode 不接受客户端对命名空间的修改操作，整个命名空间都处于只读状态。同时，Namenode 也不会向 Datanode 下发任何数据块的复制、删除指令。管理员可以通过 dfsadmin setSafemode 命令触发 Namenode 进入或者退出安全模式，同时还可以使用这个命令查询安全模式的状态。需要注意的是，刚刚启动的 Namenode 会直接自动进入安全模式，当 Namenode 中保存的满足最小副本系数的数据块达到一定的比例时，Namenode 会自动退出安全模式。而对于用户通过 dfsAdmin 方式触发 Namenode 进入安全模式的情况，则只能由管理员手动关闭安全模式，Namenode 不可以自动退出。dfsadmin setSafemode 命令正是通过调用 ClientProtocol.setSafeMode()方法实现的。setSafeMode()方法的定义如下：

```
public boolean setSafeMode(HdfsConstants.SafeModeAction action, boolean isChecked)
    throws IOException;
```

了解了安全模式之后，我们来看看必须在安全模式中才能进行的两个操作。`-saveNamespace`用于将整个命名空间保存到新的 fsimage 文件中，并且重置 editlog 文件；而`-rollEdits`则会触发重置 editlog 文件的操作，关闭当前正在写入的 editlog 文件，开启一个新的 editlog 文件（fsimage 与 editlog 文件请参考第 3 章的 fsimage 小节）。

refreshNodes()方法会触发 Namenode 刷新数据节点列表。管理员可以通过 include 文件指定可以连接到 Namenode 的数据节点列表，通过 exclude 文件指定不能连接到 Namenode 的数据节点列表。每当管理员修改了这两个配置文件后，都需要通过`-refreshNodes`选项触发 Namenode 刷新数据节点列表，这个操作会造成 Namenode 从文件系统中移除已有的数据节点，或者添加新的数据节点（请参考第 3 章的数据节点管理小节）。

finalizeUpgrade()和 rollingUpgrade()操作都是与 Namenode 升级相关的，管理员可以通过`-rollingUpgrade`选项触发 Namenode 进行升级操作。当 Namenode 成功地执行了升级操作后，管理员可以通过`-finalizeUpgrade`提交升级操作，提交升级操作会删除升级操作创建的一些临时目录，提交升级操作之后就不可以再回滚了（请参考第 4 章的 Storage 小节）。

对于其他方法，请读者参考表 1-2 中的说明，这里不再详细介绍了。

（5）快照相关操作

Hadoop 2.X 添加了新的快照特性，用户可以为 HDFS 的任意路径创建快照。快照保存了一个时间点上 HDFS 某个路径中所有数据的拷贝，快照可以将失效的集群回滚到之前一个正常的时间点上。用户可以通过 `hdfs dfs` 命令执行创建、删除以及重命名快照等操作，ClientProtocol 也定义了对应的方法来支持快照命令。

需要特别注意的是，在创建快照之前，必须先通过 `hdfs dfsadmin -allowSnapshot` 命令开启目录的快照功能，否则不可以在该目录上创建快照。表 1-3 给出了快照操作与 ClientProtocol 中相关方法的对应关系，请读者参考（快照相关内容我们会在第 3 章的快照小节中介绍）。

表 1-3 快照操作与 ClientProtocol 中相关方法的对应关系

方法名	作 用	对应的命令
createSnapshot()	创建快照	`hdfs dfs -createSnapshot`
deleteSnapshot()	删除快照	`hdfs dfs -deleteSnapshot`
renameSnapshot()	重命名快照	`hdfs dfs -renameSnapshot <path><oldName> <newName>`
allowSnapshot()	开启指定目录的快照功能。一个目录必须在开启快照功能之后才可以添加快照	`hdfs dfsadmin -allowSnapshot <path>`
disallowSnapshot()	关闭指定目录的快照功能	`hdfs dfs -deleteSnapshot <path><snapshotName>`
getSnapshotDiffReport()	获取两个快照间的不同	`hdfs snapshotDiff <path><fromSnapshot> <toSnapshot>`

（6）缓存相关操作

HDFS 2.3 版本添加了集中式缓存管理（HDFS Centralized Cache Management）功能。用户可以指定一些经常被使用的数据或者高优先级任务对应的数据，让它们常驻内存而不被淘汰到磁盘上，这对于提升 Hadoop 系统和上层应用的执行效率与实时性有很大的帮助。

这里涉及两个概念。

- **cache directive**: 表示要被缓存到内存的文件或者目录。
- **cache pool**: 用于管理一系列的 cache directive，类似于命名空间。同时使用 UNIX 风格的文件读、写、执行权限管理机制。

表 1-4 总结了缓存相关命令与 ClientProtocol 方法之间的对应关系，请读者参考（缓存相关内容我们会在第 3 章的缓存管理小节中介绍）。

表 1-4 缓存相关命令与 ClientProtocol 方法之间的对应关系

方法名	作 用	对应的命令
addCacheDirective()	添加一个缓存	`hdfs cacheadmin -addDirective -path <path> -pool <pool-name> [-force] [-replication <replication>] [-ttl <time-to-live>]`

续表

方法名	作 用	对应的命令
modifyCacheDirective()	修改缓存	-modifyDirective
removeCacheDirective()	删除缓存	`hdfs cacheadmin -removeDirective <id>`
listCacheDirectives()	列出指定路径下的所有缓存	`hdfs cacheadmin -listDirectives [-stats] [-path <path>] [-pool <pool>]`
addCachePool()	添加一个缓存池	`hdfs cacheadmin -addPool <name> [-owner <owner>] [-group <group>] [-mode <mode>] [-limit <limit>] [-maxTtl <maxTtl>]`
modifyCachePool()	修改已有缓存池的元数据	`hdfs cacheadmin -modifyPool <name> [-owner <owner>] [-group <group>] [-mode <mode>] [-limit <limit>] [-maxTtl <maxTtl>]`
removeCachePool()	删除缓存池	`hdfs cacheadmin -removePool <name>`
listCachePools()	列出已有缓存池的信息，包括用户名、用户组、权限等	`hdfs cacheadmin -listPools [-stats] [<name>]`

(7) 其他操作

安全相关以及 XAttr 相关命令，主要都是增加、删除以及 List 操作，这里就不再详细介绍了

2. ClientDatanodeProtocol

ClientDatanodeProtocol 定义了 Client 与 Datanode 之间的接口。相比 ClientProtocol，ClientDatanodeProtocol 的定义简单很多，如图 1-2 所示。

```

ClientDatanodeProtocol
- getReplicaVisibleLength(ExtendedBlock) long
- refreshNamenodes() void
- deleteBlockPool(String, boolean) void
- getBlockLocalPathInfo(ExtendedBlock, Token<BlockTokenIdentifier>) nfo
- getHdfsBlocksMetadata(String, long[], List<Token<BlockTokenIdentifier>) void
- shutdownDatanode(boolean) void
- getDatanodeInfo() DatanodeLocalInfo
- startReconfiguration() void
- getReconfigurationStatus() ReconfigurationTaskStatus

```

图 1-2 ClientDatanodeProtocol 定义

ClientDatanodeProtocol 中定义的接口可以分为两部分：一部分是支持 HDFS 文件读取操作的，例如 getReplicaVisibleLength() 以及 getBlockLocalPathInfo(); 另一部分是支持 DFSAdmin 中与数据节点管理相关的命令。下面我们就看一下图 1-2 中所示 9 个方法的具体定义。

(1) getReplicaVisibleLength()

客户端会调用 getReplicaVisibleLength() 方法从数据节点获取某个数据块副本真实的数据长度。当客户端读取一个 HDFS 文件时，需要获取这个文件对应的所有数据块的长度，用于

建立数据块的输入流，然后读取数据。但是 Namenode 元数据中文件的最后一个数据块长度与 Datanode 实际存储的可能不一致，所以客户端在创建输入流时就需要调用 `getReplicaVisibleLength()` 方法从 Datanode 获取这个数据块的真实长度。

(2) `getBlockLocalPathInfo()`

HDFS 对于本地读取，也就是 Client 和保存该数据块的 Datanode 在同一台物理机器上时，是有很多优化的。Client 会调用 `ClientProtocol.getBlockLocalPathInfo()` 方法获取指定数据块文件以及数据块校验文件在当前节点上的本地路径，然后利用这个本地路径执行本地读取操作，而不是通过流式接口执行远程读取，这样也就大大优化了读取的性能。

在 HDFS 2.6 版本中，客户端会通过调用 `DataTransferProtocol` 接口从数据节点获取数据块文件的文件描述符，然后打开并读取文件以实现短路读操作，而不是通过 `ClientDatanodeProtocol` 接口。客户端的短路读操作请参考第 5 章的文件短路读操作小节。

(3) `refreshNamenodes()`

在用户管理员命令中有一个 ``hdfs dfsadmin datanodehost:port`` 命令，用于触发指定的 Datanode 重新加载配置文件，停止服务那些已经从配置文件中删除的块池 (blockPool)，开始服务新添加的块池。块池的概念请参考第 4 章的 Datanode 逻辑结构小节。

这条命令底层就是由 `ClientDatanodeProtocol.refreshNamenodes()` 方法实现的，客户端会通过这个接口触发对应的 Datanode 执行操作。

(4) `deleteBlockPool()`

在用户管理员命令中还有一个与块池管理相关的 ``hdfs dfsadmin-deleteBlockPool datanode-host:port blockpoolId [force]`` 命令，用于从指定 Datanode 删除 blockpoolId 对应的块池，如果 force 参数被设置了，那么无论这个块池目录中有没有数据都会被强制删除；否则，只有这个块池目录为空的情况下才会被删除。需要注意的是，如果 Datanode 还在服务这个块池，这个命令的执行将会失败。要停止一个数据节点服务指定的块池，需要调用上面提到的 `refreshNamenodes()` 方法。

`deleteBlockPool()` 方法有两个参数，其中 blockpoolId 用于设置要被删除的块池 ID；force 用于设置是否强制删除。

```
void deleteBlockPool(String bpid, boolean force) throws IOException;
```

(5) `getHdfsBlocksMetadata()`

`getHdfsBlocksMetadata()` 方法主要用于获取数据块是存储在指定 Datanode 的哪个卷 (volume) 上的，这个方法主要是为了支持 `DistributedFileSystem.getFileBlockStorageLocations()` 方法。关于卷 (volume) 的定义请参考第 4 章。

(6) `shutdownDatanode()`

`shutdownDatanode()` 方法用于关闭一个数据节点，这个方法主要是为了支持管理命令 ``hdfs`

`dfsadmin-shutdownDatanode <datanode_host:ipc_port> [upgrade]`。`

(7) `getDatanodeInfo()`

`getDatanodeInfo()`方法用于获取指定 `Datanode` 的信息，这里的信息包括 `Datanode` 运行的 HDFS 版本、`Datanode` 配置的 HDFS 版本，以及 `Datanode` 的启动时间。对应于管理命令 ``hdfs dfsadmin-getDatanodeInfo``。

(8) `startReconfiguration()`

`startReconfiguration()`方法用于触发 `Datanode` 异步地从磁盘重新加载配置，并且应用该配置。这个方法用于支持管理命令 ``hdfs dfsadmin-getDatanodeInfo-reconfigstart``。

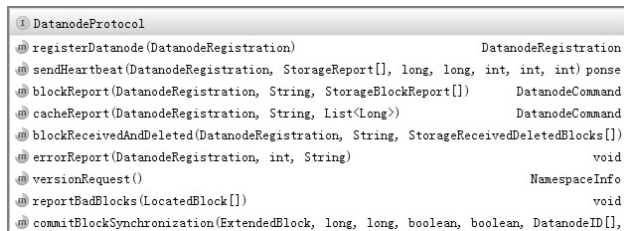
(9) `getReconfigurationStatus()`

我们知道 `startReconfiguration()`方法是异步地加载配置操作，所以 HDFS 提供了 `getReconfigurationStatus()`方法用于查询上一次触发的重新加载配置操作的运行情况。对应于管理命令 ``hdfs dfsadmin-getDatanodeInfo-reconfigstartstatus``。我们可以看到 `getReconfigurationStatus()`方法和 `startReconfiguration()`方法对应的管理命令是一样的，只不过参数不同，一个是 `start`，一个是 `status`。

3. `DatanodeProtocol`

`ClientProtocol` 和 `DatanodeProtocol` 都是由客户端发起调用的接口，下面我们介绍服务器间的接口。`DatanodeProtocol` 是 `Datanode` 与 `Namenode` 间的接口，`Datanode` 会使用这个接口与 `Namenode` 握手、注册、发送心跳、进行全量以及增量的数据块汇报。`Namenode` 会在 `Datanode` 的心跳响应中携带名字节点指令，`Datanode` 收到名字节点指令之后会执行对应的操作。要特别注意的是，`Namenode` 向 `Datanode` 下发名字节点指令是没有任何其他接口的，只会通过 `DatanodeProtocol` 的返回值来下发命令。

`DatanodeProtocol` 定义的方法如图 1-3 所示，我们可以将 `DatanodeProtocol` 定义的方法分为三种类型：`Datanode` 启动相关、心跳相关以及数据块读写相关。下面将会分别介绍这三种类型的方法，以及 `Namenode` 向 `Datanode` 下发的指令。



① DatanodeProtocol	
registerDatanode(DatanodeRegistration)	DatanodeRegistration
sendHeartbeat(DatanodeRegistration, StorageReport[], long, long, int, int, int)	ponse
blockReport(DatanodeRegistration, String, StorageBlockReport[])	DatanodeCommand
cacheReport(DatanodeRegistration, String, List<Long>)	DatanodeCommand
blockReceivedAndDeleted(DatanodeRegistration, String, StorageReceivedDeletedBlocks[])	
errorReport(DatanodeRegistration, int, String)	void
versionRequest()	NamespaceInfo
reportBadBlocks(LocatedBlock[])	void
commitBlockSynchronization(ExtendedBlock, long, long, boolean, boolean, DatanodeID[],	

图 1-3 `DatanodeProtocol` 定义

(1) `Datanode` 启动相关方法

一个完整的 `Datanode` 启动操作会与 `Namenode` 进行 4 次交互，也就是调用 4 次

DatanodeProtocol 定义的方法。首先调用 `versionRequest()` 与 Namenode 进行握手操作，然后调用 `registerDatanode()` 向 Namenode 注册当前的 Datanode，接着调用 `blockReport()` 汇报 Datanode 上存储的所有数据块，最后调用 `cacheReport()` 汇报 Datanode 缓存的所有数据块。

我们首先看一下 `versionRequest()` 方法。Datanode 启动时会首先调用 `versionRequest()` 方法与 Namenode 进行握手。这个方法的返回值是一个 `NamespaceInfo` 对象，`NamespaceInfo` 对象会封装当前 HDFS 集群的命名空间信息，包括存储系统的布局版本号 (`layoutversion`)、当前的命名空间的 ID (`namespaceId`)、集群 ID (`clusterId`)、文件系统的创建时间 (`ctime`)、构建时的 HDFS 版本号 (`buildVersion`)、块池 ID (`blockpoolId`)、当前的软件版本号 (`softwareVersion`) 等。Datanode 获取到 `NamespaceInfo` 对象后，就会比较 Datanode 当前的 HDFS 版本号和 Namenode 的 HDFS 版本号，如果 Datanode 版本与 Namenode 版本不能协同工作，则抛出异常，Datanode 也就无法注册到该 Namenode 上。如果当前 Datanode 上已经有了文件存储的目录，那么 Datanode 还会检查 Datanode 存储上的块池 ID、文件系统 ID 以及集群 ID 与 Namenode 返回的是否一致。

```
public NamespaceInfo versionRequest() throws IOException;
```

成功进行握手操作后，Datanode 会调用 `ClientProtocol.registerDatanode()` 方法向 Namenode 注册当前的 Datanode，这个方法的参数是一个 `DatanodeRegistration` 对象，它封装了 `DatanodeID`、Datanode 的存储系统的布局版本号 (`layoutversion`)、当前命名空间的 ID (`namespaceId`)、集群 ID (`clusterId`)、文件系统的创建时间 (`ctime`) 以及 Datanode 当前的软件版本号 (`softwareVersion`)。名字节点会判断 Datanode 的软件版本号与 Namenode 的软件版本号是否兼容，如果兼容则进行注册操作，并返回一个 `DatanodeRegistration` 对象供 Datanode 后续处理逻辑使用。

```
public DatanodeRegistration registerDatanode(DatanodeRegistration registration
) throws IOException;
```

Datanode 成功向 Namenode 注册之后，Datanode 会通过调用 `DatanodeProtocol.blockReport()` 方法向 Namenode 上报它管理的所有数据块的信息。这个方法需要三个参数：`DatanodeRegistration` 用于标识当前的 Datanode；`poolId` 用于标识数据块所在的块池 ID；`reports` 是一个 `StorageBlockReport` 对象的数组，每个 `StorageBlockReport` 对象都用于记录 Datanode 上一个存储空间存储的数据块。这里需要特别注意的是，上报的数据块是以长整型数组保存的，每个已经提交的数据块 (`finalized`) 以 3 个长整型来表示，每个构建中的数据块 (`under-construction`) 以 4 个长整型来表示。之所以不使用 `ExtendedBlock` 对象保存上报的数据块，是因为这样可以减少 `blockReport()` 操作所使用的内存，Namenode 接收到消息时，不需要创建大量的 `ExtendedBlock` 对象，只需要不断地从长整型数组中提取数据块即可。

```
public DatanodeCommand blockReport(DatanodeRegistration registration,
String poolId, StorageBlockReport[] reports) throws IOException;
```

Namenode 接收到 `blockReport()` 请求之后，会根据 Datanode 上报的数据块存储情况建立数据块与数据节点之间的对应关系。同时，Namenode 会在 `blockReport()` 的响应中携带名字节点指令，通知数据节点进行重新注册、发送心跳、备份或者删除 Datanode 本地磁盘上数据块

副本的操作。这些名字节点指令都是以 `DatanodeCommand` 对象封装的，我们会在 `DatanodeCommand` 小节详细介绍名字节点指令以及 `DatanodeCommand` 对象。

`blockReport()`方法只在 `Datanode` 启动时以及指定间隔时执行一次。在这里间隔是由 `dfs.blockreport.intervalMsec` 参数配置的，默认是 6 小时执行一次。`cacheReport()`方法与 `blockReport()`方法是完全一致的，只不过汇报的是当前 `Datanode` 上缓存的所有数据块。`cacheReport()`方法的定义如下：

```
public DatanodeCommand cacheReport(DatanodeRegistration registration,
    String poolId, List<Long> blockIds) throws IOException;
```

(2) 心跳相关方法

我们知道分布式系统的节点之间大多采用心跳来维护节点的健康状态。HDFS 也是一样，`Datanode` 会定期（由 `dfs.heartbeat.interval` 配置项配置，默认是 3 秒）向 `Namenode` 发送心跳，如果 `Namenode` 长时间没有接到 `Datanode` 发送的心跳，则 `Namenode` 会认为该 `Datanode` 失效。

`ClientProtocol.sendHeartbeat()`方法就是用于心跳汇报的接口，除了携带标识 `Datanode` 身份的 `DatanodeRegistration` 对象外，还包括数据节点上所有存储的状态、缓存的状态、正在写文件数据的连接数、读写数据使用的线程数等。

`sendHeartbeat()`会返回一个 `HeartbeatResponse` 对象，这个对象包含了 `Namenode` 向 `Datanode` 发送的名字节点指令，以及当前 `Namenode` 的 HA 状态。需要特别注意的是，在开启了 HA 的 HDFS 集群中，`Datanode` 是需要同时向 `Active Namenode` 以及 `Standby Namenode` 发送心跳的，不过只有 `ActiveNamenode` 才能向 `Datanode` 下发名字节点指令。

```
public HeartbeatResponse sendHeartbeat(DatanodeRegistration registration,
    StorageReport[] reports,
    long dnCacheCapacity,
    long dnCacheUsed,
    int xmitsInProgress,
    int xceiverCount,
    int failedVolumes) throws IOException;
```

(3) 数据块读写相关方法

上面两个小节我们介绍了 `Datanode` 启动时以及发送心跳时与 `Namenode` 交互的方法。这一小节将介绍 `Datanode` 在进行数据块读写操作时与 `Namenode` 交互的方法，包括 `DatanodeProtocol` 中的 `reportBadBlocks()`、`blockReceivedAndDeleted()`以及 `commitBlockSynchronization()`方法。下面我们依次介绍这三个方法的定义及使用。

`reportBadBlocks()`与 `ClientProtocol.reportBadBlocks()`方法很类似，`Datanode` 会调用这个方法向 `Namenode` 汇报损坏的数据块。`Datanode` 会在三种情况下调用这个方法：`DataBlockScanner` 线程定期扫描数据节点上存储的数据块，发现数据块的校验出现错误时；数据流管道写数据时，`Datanode` 接受了一个新的数据块，进行数据块校验操作出现错误时；进行数据块复制操作（`DataTransfer`），`Datanode` 读取本地存储的数据块时，发现本地数据块副本的长度小于 `Namenode` 记录的长度，则认为该数据块已经无效，会调用 `reportBadBlocks()`方法。

`reportBadBlocks()`方法的参数是 `LocatedBlock` 对象，这个对象描述了出现错误数据块的位置，`Namenode` 收到 `reportBadBlocks()`请求后，会下发数据块副本删除指令删除错误的数据块。

```
public void reportBadBlocks(LocatedBlock[] blocks) throws IOException;
```

`Datanode` 会定期（默认是 5 分钟，不可以配置）调用 `blockReceivedAndDeleted()`方法向 `Namenode` 汇报 `Datanode` 新接受的数据块或者删除的数据块。`Datanode` 接受一个数据块，可能是因为 `Client` 写入了新的数据块，或者从别的 `Datanode` 上复制一个数据块到当前 `Datanode`。`Datanode` 删除一个数据块，则有可能是因为该数据块的副本数量过多，`Namenode` 向当前 `Datanode` 下发了删除数据块副本的指令。我们可以把 `blockReceivedAndDeleted()`方法理解为 `blockReport()`的增量汇报，这个方法的参数包括 `DatanodeRegistration` 对象、增量汇报数据块所在的块池 ID，以及 `StorageReceivedDeletedBlocks` 对象的数组，这里的 `StorageReceivedDeletedBlocks` 对象封装了 `Datanode` 的一个数据存贮上新添加以及删除的数据块集合。`Namenode` 接受了这个请求之后，会更新它内存中数据块与数据节点的对应关系。

```
public void blockReceivedAndDeleted(DatanodeRegistration registration,
    String poolId,
    StorageReceivedDeletedBlocks[] rcvdAndDeletedBlocks)
    throws IOException;
```

`DatanodeProtocol` 中与数据块读写相关的最后一个方法是 `commitBlockSynchronization()`，这个方法用于在租约恢复操作时同步数据块的状态。在租约恢复操作时，主数据节点完成所有租约恢复协调操作后调用 `commitBlockSynchronization()`方法同步 `Datanode` 和 `Namenode` 上数据块的状态，所以 `commitBlockSynchronization()`方法包含了大量的参数。对于租约恢复，我们会在第 3 章的租约管理小节以及第 4 章的文件系统数据集小节中介绍，请读者参考这两个章节内容。

```
public void commitBlockSynchronization(ExtendedBlock block,
    long newgenerationstamp, long newlength,
    boolean closeFile, boolean deleteblock, DatanodeID[] newtargets,
    String[] newtargetstorages) throws IOException;
```

（4）其他方法

`DataProtocol` 中最后一个方法是 `errorReport()`，该方法用于向名字节点上报运行过程中发生的一些状况，如磁盘不可用等，这个方法在调试时非常有用。由于这个方法并不涉及具体的逻辑，我们就不再详细介绍了。

（5）`DatanodeCommand`

通过前面几个小节的介绍我们知道，`sendHeartbeat()`、`blockReport()`以及 `cacheReport()`方法的返回值都会携带 `Namenode` 向 `Datanode` 下发的名字节点指令。在 HDFS 中，使用 `DatanodeCommand` 类描述 `Namenode` 向 `Datanode` 发出的名字节点指令。`DatanodeCommand` 类以及它的子类结构如图 1-4 所示。

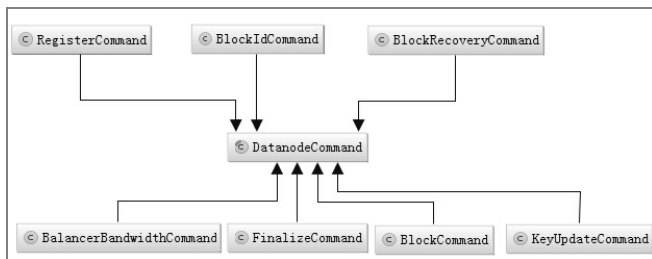


图 1-4 DatanodeCommand 类结构

DatanodeCommand 是所有名字节点的基类，它一共有 7 个子类，但在 DatanodeProtocol 中一共定义了 10 个名字节点指令，每个指令都有一个唯一的编号与之对应，代码如下：

```

public interface DatanodeProtocol {
    // ...
    final static int DNA_UNKNOWN = 0;    // 未定义
    final static int DNA_TRANSFER = 1;   // 数据块复制
    final static int DNA_INVALIDATE = 2; // 数据块删除
    final static int DNA_SHUTDOWN = 3;   // 关闭数据节点
    final static int DNA_REGISTER = 4;   // 重新注册数据节点
    final static int DNA_FINALIZE = 5;   // 提交上一次升级
    final static int DNA_RECOVERBLOCK = 6; // 数据块恢复
    final static int DNA_ACCESSKEYUPDATE = 7; // 安全相关
    final static int DNA_BALANCERBANDWIDTHUPDATE = 8; // 更新平衡器带宽
    final static int DNA_CACHE = 9;     // 缓存数据块
    final static int DNA_UNCACHE = 10;  // 解缓存数据块
    // ...
}

```

可以看到，DatanodeProtocol 中定义的指令类型数量和 DatanodeCommand 子类的数量并不一致。这是因为指令编号 DNA_SHUTDOWN 已经废弃不用了，Datanode 接收到 DNA_SHUTDOWN 指令后会直接抛出 UnsupportedOperationException 异常。关闭 Datanode 是通过调用 ClientDatanodeProtocol.shutdownDatanode() 方法来触发的。

这里同时要注意的是，DNA_TRANSFER、DNA_RECOVERBLOCK 以及 DNA_INVALIDATE 都是通过 BlockCommand 子类来封装的，只不过参数不同。DNA_TRANSFER 指令用于触发数据节点的数据块复制操作，当 HDFS 系统中某个数据块的副本数小于配置的副本系数时，Namenode 会通过 DNA_TRANSFER 指令通知某个拥有这个数据块副本的 Datanode 将该数据块复制到其他数据节点上。DNA_INVALIDATE 用于通知 Datanode 删除数据节点上的指定数据块，这是因为 Namenode 发现了某个数据块的副本数已经超过了配置的副本系数，这时 Namenode 会通知某个数据节点删除这个数据节点上多余的数据块副本。当客户端在写文件时发生异常退出，会造成数据流管道中不同数据节点上数据块状态的不一致，这时 Namenode 会从数据流管道中选出一个数据节点作为主恢复节点，协调数据流管道中的其他数据节点进行租约恢复操作，以同步这个数据块的状态。此时 Namenode 就会向这个数据节点下发 DNA_RECOVERBLOCK 指令，通知数据节点开始租约恢复操作。

至于 `DatanodeCommand` 中其他类的作用都比较简单，这里就不再单独介绍了。

4. InterDatanodeProtocol

介绍完了 `Datanode` 与 `Namenode` 之间的接口，我们来介绍 `InterDatanodeProtocol`——`Datanode` 与 `Datanode` 之间的接口。`InterDatanodeProtocol` 接口主要用于租约恢复操作，如图 1-5 所示，`InterDatanodeProtocol` 只有 `initReplicaRecovery()` 和 `updateReplicaUnderRecovery()` 两个方法。

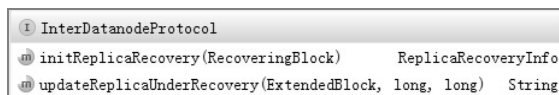


图 1-5 `InterDatanodeProtocol` 接口

客户端打开一个文件进行写操作时，首先要获取这个文件的租约，并且还需要定期更新租约。当 `Namenode` 的租约监控线程发现某个 HDFS 文件租约长期没有更新时，就会认为写这个文件的客户端发生异常，这时 `Namenode` 就需要触发租约恢复操作——同步数据流管道中所有 `Datanode` 上该文件数据块的状态，并强制关闭这个文件。

租约恢复的控制并不是由 `Namenode` 负责的，而是 `Namenode` 从数据流管道中选出一个主恢复节点，然后通过下发 `DatanodeCommand` 的恢复指令触发这个数据节点控制租约恢复操作，也就是由这个主恢复节点协调整个租约恢复操作的过程。主恢复节点会调用 `InterDatanodeProtocol` 接口来指挥数据流管道的其他数据节点进行租约恢复。租约恢复操作其实很简单，就是将数据流管道中所有数据节点上保存的同一个数据块状态（时间戳和数据块长度）同步一致。当成功完成租约恢复后，主恢复节点会调用 `DatanodeProtocol.commitBlockSynchronization()` 方法同步名字节点上该数据块的时间戳和数据块长度，保持名字节点和数据节点的一致。

由于数据流管道中同一个数据块状态（长度和时间戳）在不同的 `Datanode` 上可能是不一致的，所以主恢复节点会首先调用 `InterDatanodeProtocol.initReplicaRecovery()` 方法获取数据流管道中所有数据节点上保存的指定数据块的状态，这里的数据块状态使用 `ReplicaRecoveryInfo` 类封装。主恢复节点会根据收集到的这些状态，确定一个当前数据块的新长度，并且使用 `Namenode` 下发的 `recoverId` 作为数据块的新时间戳。

```
ReplicaRecoveryInfo initReplicaRecovery(RecoveringBlock rBlock)
throws IOException;
```

主恢复节点计算出数据块的新长度后，就会调用 `InterDatanodeProtocol.updateReplicaUnderRecovery()` 方法将数据流管道中所有节点上该数据块的长度同步为新的长度，将数据块的时间戳同步为新的时间戳。

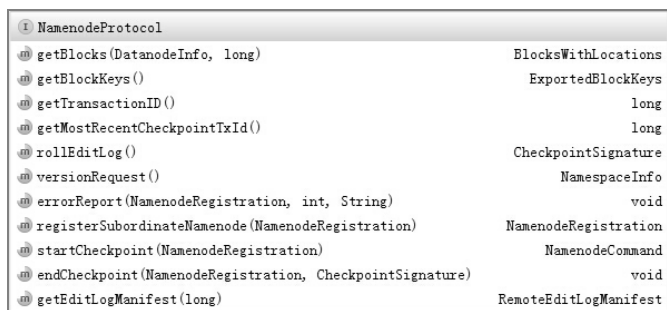
```
String updateReplicaUnderRecovery(ExtendedBlock oldBlock, long recoveryId,
    long newLength) throws IOException;
```

当完成了所有的同步操作后，主恢复节点就可以调用 `DatanodeProtocol.commitBlockSynchronization()` 将 `Namenode` 上该数据块的长度和时间戳同步为新的长度和时间戳，这样

Datanode 和 Namenode 的数据也就一致了。

5. NamenodeProtocol

NamenodeProtocol 定义了第二名字节点与名字节点之间的接口。由于在 Hadoop 2.X 架构中第二名字节点的功能已经完全被 Standby 节点所取代，这个接口我们就不再详细介绍了。图 1-6 给出了 NamenodeProtocol 定义的所有方法，读者可以参考这个图读取相关的代码。



1 NamenodeProtocol	
getBlocks(DatanodeInfo, long)	BlocksWithLocations
getBlockKeys()	ExportedBlockKeys
getTransactionID()	long
getMostRecentCheckpointTxId()	long
rollEditLog()	CheckpointSignature
versionRequest()	NamespaceInfo
errorReport(NamenodeRegistration, int, String)	void
registerSubordinateNamenode(NamenodeRegistration)	NamenodeRegistration
startCheckpoint(NamenodeRegistration)	NamenodeCommand
endCheckpoint(NamenodeRegistration, CheckpointSignature)	void
getEditLogManifest(long)	RemoteEditLogManifest

图 1-6 NamenodeProtocol 定义

1.2.2 流式接口

HDFS 除了定义 RPC 调用接口外，还定义了流式接口，流式接口是 HDFS 中基于 TCP 或者 HTTP 实现的接口。在 HDFS 中，流式接口包括了基于 TCP 的 DataTransferProtocol 接口，以及 HA 架构中 Active Namenode 和 Standby Namenode 之间的 HTTP 接口。

1. DataTransferProtocol

DataTransferProtocol 是用来描述写入或者读出 Datanode 上数据的基于 TCP 的流式接口，HDFS 客户端与数据节点以及数据节点与数据节点之间的数据块传输就是基于 DataTransferProtocol 接口实现的。HDFS 没有采用 Hadoop RPC 来实现 HDFS 文件的读写功能，是因为 Hadoop RPC 框架的效率目前还不足以支撑超大文件的读写，而使用基于 TCP 的流式接口有利于批量处理数据，同时提高了数据的吞吐量

图 1-7 给出了 DataTransferProtocol 的方法定义，其中最重要的方法就是 readBlock()和 writeBlock()。

- readBlock(): 从当前 Datanode 读取指定的数据块。
- writeBlock(): 将指定数据块写入数据流管道 (pipeLine) 中。
- transferBlock(): 将指定数据块复制 (transfer) 到另一个 Datanode 上。数据块复制操作是指数据流管道中的数据节点出现故障，需要用新的数据节点替换异常的数据节点时，DFSClient 会调用这个方法将数据流管道中异常数据节点上已经写入的数据块复制到新添加的数据节点上。

DataTransferProtocol	
readBlock(ExtendedBlock, Token<BlockTokenIdentifier>, String, long, long, boolean, Caching	void
writeBlock(ExtendedBlock, StorageType, Token<BlockTokenIdentifier>, String, DatanodeInfo[], StorageType[], DatanodeIn	
transferBlock(ExtendedBlock, Token<BlockTokenIdentifier>, String, DatanodeInfo[], Stora	void
requestShortCircuitFds(ExtendedBlock, Token<BlockTokenIdentifier>, Slot)	void
releaseShortCircuitFds(Slot)	void
requestShortCircuitShm(Stri	void
replaceBlock(ExtendedBlock, StorageType, Token<BlockTokenIdentifier>, String, Data	void
copyBlock(ExtendedBlock, Token<BlockTokenIdentif	void
blockChecksum(ExtendedBlock, Token<BlockTokenIdentif	void

图 1-7 DataTransferProtocol 接口

- **replaceBlock()**: 将从源 Datanode 复制来的数据块写入本地 Datanode。写成功后通知 NameNode，并且删除源 Datanode 上的数据块。这个方法主要用在数据块平衡操作（balancing）的场景下。
- **copyBlock()**: 复制当前 Datanode 上的数据块。这个方法主要用在数据块平衡操作的场景下。
- **blockChecksum()**: 获取指定数据块的校验值。
- **requestShortCircuitFds()**: 获取一个短路（short circuit）读取数据块的文件描述符（请参考第 5 章的文件短路读操作小节）。
- **releaseShortCircuitFds()**: 释放一个短路读取数据块的文件描述符。
- **requestShortCircuitShm()**: 获取保存短路读取数据块的共享内存。

DataTransferProtocol 接口调用并没有使用 Hadoop RPC 框架提供的功能，而是定义了用于发送 DataTransferProtocol 请求的 Sender 类，以及用于响应 DataTransferProtocol 请求的 Receiver 类，Sender 类和 Receiver 类都实现了 DataTransferProtocol 接口。图 1-8 给出了 DataTransferProtocol 接口的一个调用示例。我们假设 DFSCClient 发起了一个 DataTransferProtocol.readBlock()操作，那么 DFSCClient 会调用 Sender 将这个请求序列化，并传输给远端的 Receiver。远端的 Receiver 接收到这个请求后，会反序列化请求，然后调用代码执行读取操作。

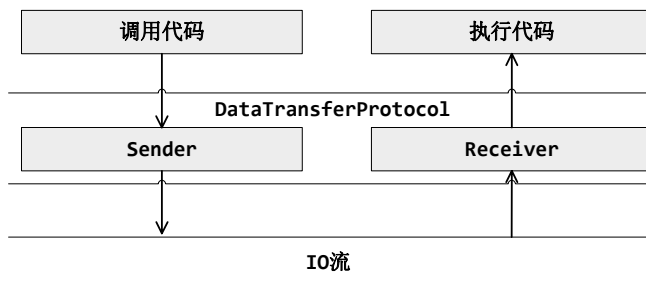


图 1-8 DataTransferProtocol 调用示例

DataTransferProtocol 接口的具体实现我们会在第 4 章的流式接口小节中详细介绍，请读者参考该小节内容学习。

2. Active Namenode 和 Standby Namenode 间的 HTTP 接口

Namenode 会定期将文件系统的命名空间（文件目录树、文件/目录元信息）保存到一个名叫 `fsimage` 的文件中，以防止 Namenode 掉电或者进程崩溃。但如果 Namenode 实时地将内存中的命名空间同步到 `fsimage` 文件中，将会非常地消耗资源且造成 Namenode 运行缓慢。所以 Namenode 会先将命名空间的修改操作保存在 `editlog` 文件中，然后定期合并 `fsimage` 和 `editlog` 文件。

合并 `fsimage` 和 `editlog` 文件是非常耗费资源的，所以在 Hadoop 2.X 版本之前，HDFS 引入了一个第二名字节点专门负责合并 `fsimage` 和 `editlog` 文件。而在 Hadoop 2.X 版本中，由于 Standby Namenode 会不断地将读入的 `editlog` 文件与当前的命名空间合并，从而始终保持着一个最新版本的命名空间，所以 Standby Namenode 只需定期将自己的命名空间写入一个新的 `fsimage` 文件，并通过 HTTP 协议将这个 `fsimage` 文件传回 Active Namenode 即可。

Active Namenode 和 Standby Namenode 之间的 HTTP 接口就是用来传输这个新的 `fsimage` 文件的。Standby Namenode 成功地将自己的命名空间写入新的 `fsimage` 文件后，就会向 Active Namenode 的 `ImageServlet` 发送 HTTP GET 请求 `/getimage?putimage=1`。这个请求的 URL 中包括了新的 `fsimage` 文件的事务 ID，以及 Standby Namenode 用于下载的端口和 IP 地址。Active Namenode 接收到这个请求后，会根据 Standby Namenode 提供的信息向 Standby Namenode 的 `ImageServlet` 发起 HTTP GET 请求以下载 `fsimage` 文件。

Active Namenode 和 Standby Namenode 的内容请读者参考第 3 章的 `FSImage` 以及 HA 小节。

1.3 HDFS 主要流程

在介绍完 HDFS 各种节点间的接口后，这一节我们将重点介绍 HDFS 的几个典型流程：客户端读 HDFS 文件流程、客户端写 HDFS 文件流程、客户端追加写 HDFS 文件流程、数据节点与名字节点交互流程以及 HDFS HA 切换流程等。

1.3.1 HDFS 客户端读流程

图 1-9 给出了 HDFS 客户端读取一个 HDFS 文件的流程，可以分为如下几个步骤。

- 打开 HDFS 文件：HDFS 客户端首先调用 `DistributedFileSystem.open()` 方法打开 HDFS 文件，这个方法在底层会调用 `ClientProtocol.open()` 方法，该方法会返回一个 `HdfsDataInputStream` 对象用于读取数据块。`HdfsDataInputStream` 其实是一个 `DFSInputStream` 的装饰类，真正进行数据块读取操作的是 `DFSInputStream` 对象。
- 从 Namenode 获取 Datanode 地址：在 `DFSInputStream` 的构造方法中，会调用 `ClientProtocol.getBlockLocations()` 方法向名字节点获取该 HDFS 文件起始位置数据块的位置信息。Namenode 返回的数据块的存储位置是按照与客户端的距离远近排序

的，所以 `DFSInputStream` 可以选择一个最优的 `Datanode` 节点，然后与这个节点建立数据连接读取数据块。

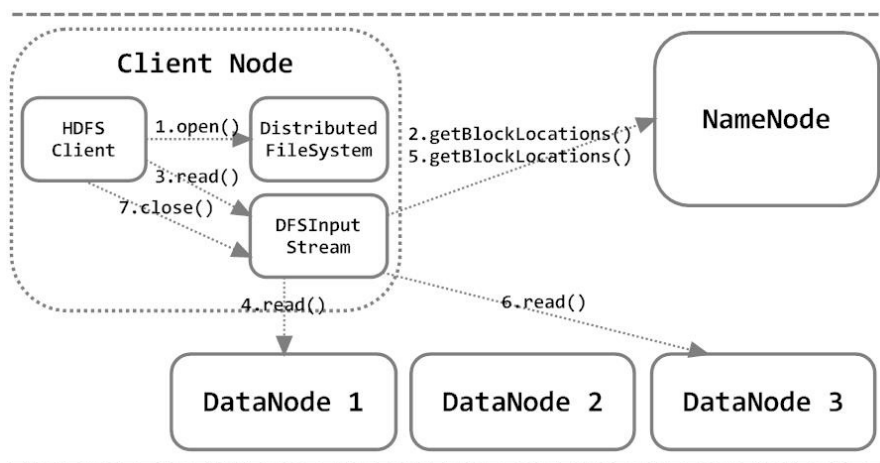


图 1-9 HDFS 读流程

- 连接到 `Datanode` 读取数据块：HDFS 客户端通过调用 `DFSInputStream.read()` 方法从这个最优的 `Datanode` 读取数据块，数据会以数据包（packet）为单位从数据节点通过流式接口传送到客户端。当达到一个数据块的末尾时，`DFSInputStream` 就会再次调用 `ClientProtocol.getBlockLocations()` 获取文件下一个数据块的位置信息，并建立和这个新的数据块的最优节点之间的连接，然后 HDFS 客户端就可以继续读取数据块了。
- 关闭输入流：当客户端成功完成文件读取后，会通过 `HdfsDataInputStream.close()` 方法关闭输入流。

客户端读取数据块时，很有可能存储这个数据块的数据节点出现异常，也就是无法读取数据。出现这种情况时，`DFSInputStream` 会切换到另一个保存了这个数据块副本的数据节点，然后读取数据。同时需要注意的是，数据块的应答包中不仅包含了数据，还包含了校验值。HDFS 客户端接收到数据应答包时，会对数据进行校验，如果出现校验错误，也就是数据节点上的这个数据块副本出现了损坏，HDFS 客户端就会通过 `ClientProtocol.reportBadBlocks()` 向 `Namenode` 汇报这个损坏的数据块副本，同时 `DFSInputStream` 会尝试从其他的数据节点读取这个数据块。

HDFS 客户端发起读操作请求以及 `HdfsDataInputStream` 输入流的实现会在第 5 章的文件读操作与输入流小节中介绍，`Datanode` 响应读数据请求的实现会在第 4 章的流式接口小节中介绍，`Namenode` 响应 HDFS 客户端 RPC 请求的实现会在第 3 章的 `ClientProtocol` 实现小节中介绍。

1.3.2 HDFS 客户端写流程

在介绍完客户端读文件流程后，下面我们学习客户端写文件流程。图 1-10 给出了 HDFS 客户端写入一个 HDFS 文件的流程。

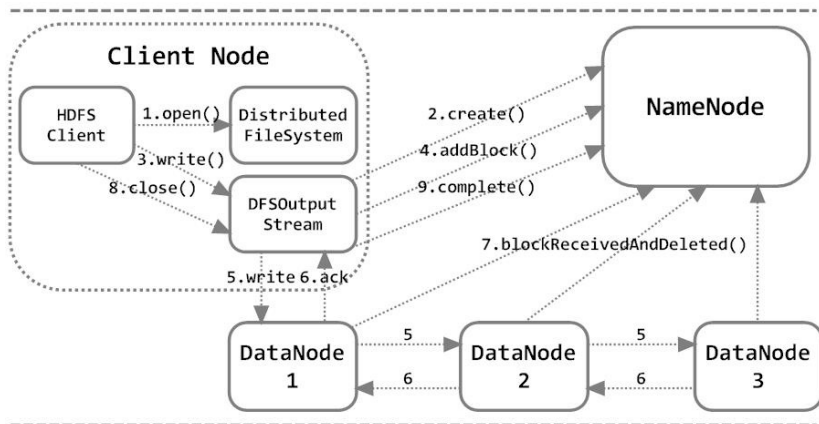


图 1-10 HDFS 写流程

- **创建文件:** HDFS 客户端写一个新的文件时，会首先调用 `DistributedFileSystem.create()` 方法在 HDFS 文件系统中创建一个新的空文件。这个方法在底层会通过调用 `ClientProtocol.create()` 方法通知 Namenode 执行对应的操作，Namenode 会首先在文件系统目录树中的指定路径下添加一个新的文件，然后将创建新文件的操作记录到 `editlog` 中。完成 `ClientProtocol.create()` 调用后，`DistributedFileSystem.create()` 方法就会返回一个 `HdfsDataOutputStream` 对象，这个对象在底层包装了一个 `DFSOutputStream` 对象，真正执行写数据操作的其实是 `DFSOutputStream` 对象。
- **建立数据流管道:** 获取了 `DFSOutputStream` 对象后，HDFS 客户端就可以调用 `DFSOutputStream.write()` 方法来写数据了。由于 `DistributedFileSystem.create()` 方法只是在文件系统目录树中创建了一个空文件，并没有申请任何数据块，所以 `DFSOutputStream` 会首先调用 `ClientProtocol.addBlock()` 向 Namenode 申请一个新的空数据块，`addBlock()` 方法会返回一个 `LocatedBlock` 对象，这个对象保存了存储这个数据块的所有数据节点的位置信息。获得了数据流管道中所有数据节点的信息后，`DFSOutputStream` 就可以建立数据流管道写数据块了。
- **通过数据流管道写入数据:** 成功地建立数据流管道后，HDFS 客户端就可以向数据流管道写数据了。写入 `DFSOutputStream` 中的数据会先被缓存在数据流中，之后这些数据会被切分成一个个数据包 (packet) 通过数据流管道发送到所有数据节点。这里的每个数据包都会按照图 1-10 所示，通过数据流管道依次写入数据节点的本地存储。每个数据包都有个确认包，确认包会逆序通过数据流管道回到输出流。输出流在确认了所有数据节点已经写入这个数据包之后，就会从对应的缓存队列删除这个数据包。当客户端写满一个数据块之后，会调用 `addBlock()` 申请一个新的数据块，

然后循环执行上述操作。

- 关闭输入流并提交文件：当 HDFS 客户端完成了整个文件中所有数据块的写操作之后，就可以调用 `close()` 方法关闭输出流，并调用 `ClientProtocol.complete()` 方法通知 Namenode 提交这个文件中的所有数据块，也就完成了整个文件的写入流程。

对于 Datanode，当 Datanode 成功地接受一个新的数据块时，Datanode 会通过 `DatanodeProtocol.blockReceivedAndDeleted()` 方法向 Namenode 汇报，Namenode 会更新内存中的数据块与数据节点的对应关系。

如果客户端在写文件时，数据流管道中的数据节点出现故障，则输出流会进行如下操作来进行故障恢复。

- 输出流中缓存的没有确认的数据包会重新加入发送队列，这种机制确保了数据节点出现故障时不会丢失任何数据，所有的数据都是经过确认的。但是输出流会通过调用 `ClientProtocol.updateBlockForPipeline()` 方法为数据块申请一个新的时间戳，然后使用这个新的时间戳重新建立数据流管道。这种机制保证了故障 Datanode 上的数据块的时间戳会过期，然后在故障恢复之后，由于数据块的时间戳与 Namenode 元数据中的不匹配而被删除，保证了集群中所有数据块的正确性。
- 故障数据节点会从输入流管道中删除，然后输出流会通过调用 `ClientProtocol.getAdditionalDatanode()` 方法通知 Namenode 分配新的数据节点到数据流管道中。接下来输出流会将新分配的 Datanode 添加到数据流管道中，并使用新的时间戳重新建立数据流管道。由于新添加的数据节点上并没有存储这个新的数据块，这时 HDFS 客户端会通过 `DataTransferProtocol` 通知数据流管道中的一个 Datanode 复制这个数据块到新的 Datanode 上。
- 数据流管道重新建立之后，输出流会调用 `ClientProtocol.updatePipeline()` 更新 Namenode 中的元数据。至此，一个完整的故障恢复流程就完成了，客户端可以正常完成后续的写操作了。

HDFS 客户端发起写操作请求以及 `HdfsDataOutputStream` 输出流的实现会在第 5 章的文件写操作与输出流小节中介绍，Datanode 响应写数据请求的实现会在第 4 章的流式接口小节中介绍，Namenode 响应 HDFS 客户端 RPC 请求的实现会在第 3 章的 `ClientProtocol` 实现小节中介绍。

1.3.3 HDFS 客户端追加写流程

客户端除了可以执行写入新的 HDFS 文件操作外，还可以打开一个已有文件并执行追加写操作。

- 打开已有的 HDFS 文件：客户端调用 `DistributedFileSystem.append()` 方法打开一个已有的 HDFS 文件，`append()` 方法首先会调用 `ClientProtocol.append()` 方法获取文件最后一个数据块的位置信息，如果文件的最后一个数据块已经写满则返回 `null`。然后 `append()` 方法会调用 `DFSOutputStream.newStreamForAppend()` 方法创建到这个数据块

的 `DFSOutputStream` 输出流对象，获取文件租约，并将新构建的 `DFSOutputStream` 方法包装为 `HdfsDataOutputStream` 对象，最后返回。

- 建立数据流管道：`DFSOutputStream` 类的构造方法会判断文件最后一个数据块是否已经写满，如果没有写满，则根据 `ClientProtocol.append()` 方法返回的该数据块的位置信息建立到该数据块的数据流管道；如果写满了，则调用 `ClientProtocol.addBlock()` 向 `Namenode` 申请一个新的空数据块之后建立数据流管道。
- 通过数据流管道写入数据：成功地建立数据流管道后，HDFS 客户端就可以向数据流管道写数据了，这部分内容与上节描述的写 HDFS 文件流程类似。
- 关闭输入流并提交文件：与上节描述的写 HDFS 文件流程类似，当 HDFS 客户端完成了追加写操作后，需要调用 `close()` 方法关闭输出流，并调用 `ClientProtocol.complete()` 方法通知 `Namenode` 提交这个文件中的所有数据块。

可以看到，HDFS 客户端追加写流程与写流程是很类似的，只不过在初始建立数据流管道时有些不同，请读者们注意。HDFS 客户端发起追加写操作请求的部分会在第 5 章的文件写操作与输出流小节中介绍，`Datanode` 响应写数据请求的实现会在第 4 章的流式接口小节中介绍，`Namenode` 响应 HDFS 客户端 RPC 请求的实现会在第 3 章的 `ClientProtocol` 实现小节中介绍。

1.3.4 Datanode 启动、心跳以及执行名字节点指令流程

上面介绍的三个流程主要是 HDFS 客户端发起的流程，这一节我们介绍 `Datanode` 与 `Namenode` 的交互流程，如图 1-11 所示，包括 `Datanode` 启动时的注册流程、心跳流程、数据块汇报以及增量汇报等流程。

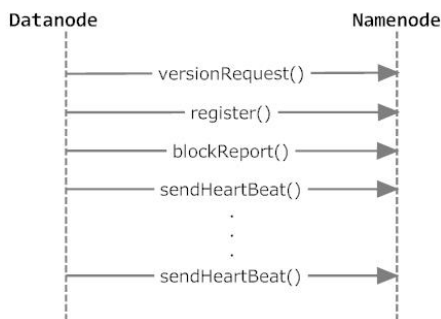


图 1-11 Datanode 启动、心跳以及执行名字节点指令流程

`Datanode` 启动后与 `Namenode` 的交互主要包括三个部分：①握手；②注册；③块汇报以及缓存汇报。

- `Datanode` 启动时会首先通过 `DatanodeProtocol.versionRequest()` 获取 `Namenode` 的版本号以及存储信息等，然后 `Datanode` 会对 `Namenode` 的当前软件版本号和 `Datanode` 的当前软件、版本号进行比较，确保它们是一致的。

- 成功地完成握手操作后, `Datanode` 会通过 `DatanodeProtocol.register()` 方法向 `Namenode` 注册。`Namenode` 接收到注册请求后, 会判断当前 `Datanode` 的配置是否属于这个集群, 它们之间的版本号是否一致。
- 注册成功之后, `Datanode` 就需要将本地存储的所有数据块以及缓存的数据块上报到 `Namenode`, `Namenode` 会利用这些信息重新建立内存中数据块与 `Datanode` 之间的对应关系。

至此, `Datanode` 就完成了启动的所有操作, 之后就可以正常对外服务了。

`Datanode` 成功启动之后, 需要定期向 `Namenode` 发送心跳, 让 `Namenode` 知道当前 `Datanode` 处于活动状态能够对外服务。`Namenode` 会在 `Datanode` 的心跳响应中携带名字节点指令, 指导 `Datanode` 进行数据块的复制、删除以及恢复等操作, 具体可以执行哪些指令请参考本章的 `DatanodeCommand` 小节。

当 `Datanode` 成功地添加了一个新的数据块或者删除了一个已有的数据块时, 需要通过 `DatanodeProtocol.blockReceivedAndDeleted()` 方法向 `Namenode` 汇报。`Namenode` 接收到这个汇报之后, 会更新 `Namenode` 内存中数据块与数据节点之间的对应关系。

`Datanode` 向 `Namenode` 发起交互请求的内容会在第4章的 `BlockPoolManager` 小节中介绍, 而 `Namenode` 对 `Datanode` 请求的响应则会在第3章的数据块管理以及数据节点管理小节中介绍。

1.3.5 HA 切换流程

在 Hadoop 2.X 之前, `Namenode` 是 HDFS 集群中可能发生单点故障的节点, 即每个 HDFS 集群中只有一个 `Namenode`, 一旦这个节点不可用, 整个 HDFS 集群就将处于不可用状态。

HDFS 的高可用 (High Availability, HA) 方案就是为了解决上述问题而产生的。图 1-12 给出了一个 HA HDFS 集群, 在 HA HDFS 集群中会同时运行两个 `Namenode`, 一个作为活动的 (Active) `Namenode`, 一个作为备份的 (Standby) `Namenode`。Active `Namenode` 的命名空间与 Standby `Namenode` 是实时同步的, 所以当 Active `Namenode` 发生故障而停止服务时, Standby `Namenode` 可以立即切换为活动状态, 而不影响 HDFS 集群的服务。

为了使 Standby 节点与 Active 节点的状态能够同步一致, 就要求两个 `Namenode` 的命名空间一致并且数据块与数据节点之间的对应关系一致。对于命名空间的一致性, 两个节点都需要与一组独立运行的节点 (JournalNodes, JNS) 通信, 当 Active `Namenode` 执行了修改命名空间的操作时, 它会定期将执行的操作记录在 `editlog` 中, 并写入 JNS 的多数节点中。而 Standby `Namenode` 会一直监听 JNS 上 `editlog` 的变化, 如果发现 `editlog` 有改动, Standby `Namenode` 就会读取 `editlog` 并与当前的命名空间合并。当发生了错误切换时, Standby 节点会先保证已经从 JNS 上读取了所有的 `editlog` 并与命名空间合并, 然后才会从 Standby 状态切换为 Active 状态。通过这种机制, 保证了 Active `Namenode` 与 Standby `Namenode` 之间命名空间状态的一致性。而对于数据块与数据节点对应关系的一致性, 则要求 HDFS 集群中的所有

Datanode 同时向这两个 Namenode 发送心跳以及块汇报信息,这样 Active Namenode 和 Standby Namenode 的数据块与数据节点之间的对应关系也就完全同步了。一旦发生故障,就可以马上切换,也就是热备。

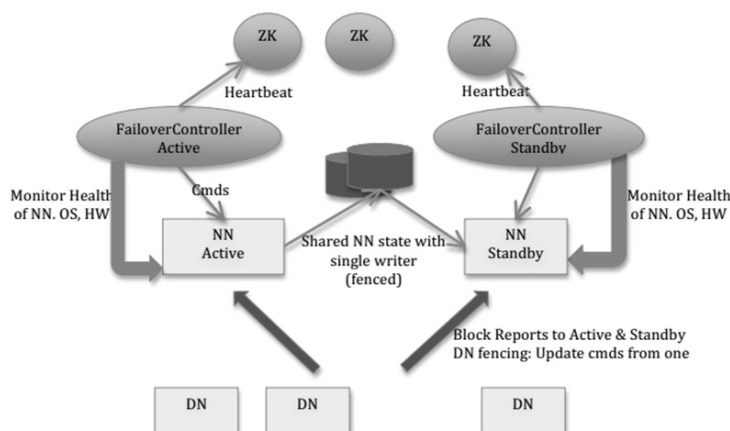


图 1-12 HA 切换流程

HDFS 提供了 HA 管理命令 (DFSHAAdmin) 使得管理员可以手动执行主备切换,同时还提供了自动 Failover 机制,该机制依赖于两个新增的网元:一个是 ZooKeeper 集群;一个是 ZKFailoverController (org.apache.hadoop.ha.ZKFailoverController)。ZKFailoverController 会实时监控 Namenode 的 HA 状态,如果 Active Namenode 处于不可服务状态,那么 ZKFailoverController 会自动触发主备切换操作,无须管理员执行任何命令。

我们会在第 3 章的 HDFS High Availability 小节中重点介绍 HDFS HA 架构的实现,请读者参考该小节内容。

第 2 章 Hadoop RPC

Hadoop 作为分布式存储系统，各个节点之间的通信和交互是必不可少的，所以需要实现一套节点间的通信交互机制。RPC（Remote Procedure Call Protocol，远程过程调用协议）允许本地程序像调用本地方法一样调用远程机器上应用程序提供的服务，所以 Hadoop 实现了一套自己的 RPC 框架。HadoopRPC 框架并没有使用 JDK 自带的 RMI（Remote Method Invocation，远程方法调用），而是基于 IPC（Inter-Process Communications，进程间通信）模型实现了一套高效的轻量级 RPC 框架，这套 RPC 框架底层采用了 JavaNIO、Java 动态代理以及 protobuf 等基础技术。

本章将介绍 HadoopRPC 框架的实现细节，但由于篇幅原因不会介绍底层依赖的 NIO、动态代理以及 protobuf 等技术，请读者自行查阅这部分内容。

2.1 概述

在介绍 Hadoop RPC 实现细节之前，有必要简要地介绍一下 Hadoop RPC 框架的总体结构以及使用方式。

2.1.1 RPC 框架概述

RPC（Remote Procedure Call Protocol，远程过程调用协议）是一种通过网络调用远程计算机服务的协议。RPC 协议假定存在某些网络传输协议，如 TCP 或 UDP，RPC 会使用这些协议传递 RPC 请求以及响应信息。RPC 协议使得分布式程序的开发更加容易，因此很受欢迎。

RPC 采用客户端/服务器模式，请求程序就是一个客户端，而服务提供程序就是一个服务器。客户端首先会发送一个有参数的调用请求到服务器，然后等待服务器发回响应信息。在服务器端，服务提供程序会保持睡眠状态直到有调用请求到达为止。当一个调用请求到达后，服务提供程序会执行调用请求，计算结果，向客户端发送响应信息，然后等待下一个调用请求。最后，客户端成功地接收服务器发回的响应信息，一个远程调用结束。

图 2-1 给出了 RPC 框架的结构，主要包括以下几部分。

- 通信模块：传输 RPC 请求和响应的网络通信模块，可以基于 TCP 协议，也可以基于 UDP 协议，可以是同步的，也可以是异步的。

- 客户端 Stub 程序：服务器和客户端都包括 Stub 程序。在客户端，Stub 程序表现得就像本地程序一样，但底层却会将调用请求和参数序列化并通过通信模块发送给服务器。之后 Stub 程序会等待服务器的响应信息，将响应信息反序列化并返回给请求程序。
- 服务器端 Stub 程序：在服务器端，Stub 程序会将远程客户端发送的调用请求和参数反序列化，根据调用信息触发对应的服务程序，然后将服务程序返回的响应信息序列化并发回客户端。
- 请求程序：请求程序会像调用本地方法一样调用客户端 Stub 程序，然后接收 Stub 程序返回的响应信息。
- 服务程序：服务器会接收来自 Stub 程序的调用请求，执行对应的逻辑并返回执行结果。

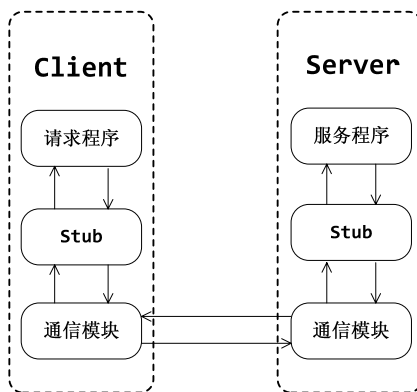


图 2-1 RPC 框架结构图

2.1.2 Hadoop RPC 框架概述

Hadoop RPC 框架与其他 RPC 框架一样实现了图 2-1 所示的结构，本节介绍 RPC 框架中各个模块在 Hadoop RPC 中的实现。

1. 通信模块

Hadoop 实现了 `org.apache.hadoop.ipc.Client` 类（之后简称为 Client 类）以及 `org.apache.hadoop.ipc.Server` 类（之后简称为 Server 类）提供的基于 TCP/IP Socket 的网络通信功能，客户端可以通过 Client 类将序列化的请求发送到远程服务器，服务器会通过 Server 类接收来自客户端的请求。

客户端在发送请求到远程服务器前需要先将请求序列化，然后调用 `Client.call()` 方法发送这个请求到远程服务器。为了使 RPC 机制更加健壮，Hadoop RPC 允许客户端配置使用不同的序列化框架序列化 RPC 请求（例如 `protobuf`、`avro`），这就要求 `Client.call()` 方法的定义更加通用，也就是 `Client.call()` 方法可以发送任意序列化框架产生的 RPC 请求。所以 `Client.call()` 方法的定义如下代码所示，`rpcKind` 参数用于描述 RPC 请求的序列化工具类型，`rpcRequest` 参数则用于记录序列化后的 RPC 请求。这里需要注意的是，`rpcRequest` 是 `Writable` 类型的

(Hadoop 框架自己定义的序列化类型), 这就要求客户端 Stub 程序将 RPC 请求序列化后包装成 Writable 类型, 所以 WritableRpcEngine 定义了 Invocation 类包装 Writable 序列化的 RPC 请求, 而 ProtobufRpcEngine 则定义了 RpcRequestWrapper 包装 protobuf 序列化的 RPC 请求。这里的 Client 类、WritableRpcEngine 类以及 ProtobufRpcEngine 类的实现会在 Hadoop RPC 实现小节中介绍。

```
public Writable call(RPC.RpcKind rpcKind, Writable rpcRequest,
    ConnectionId remoteId, AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    // ...
}
```

再看服务器端, 为了提高性能, Server 类采用了 Java NIO 提供的基于 Reactor 设计模式的事件驱动 I/O 模型, 当 Server 完整地从网络接收一个 RPC 请求后, 会调用 call() 方法响应这个请求, 如下代码所示, Server.call() 方法的定义与 Client.call() 很相似。Server 的实现也会在 Hadoop RPC 实现小节中介绍。

```
public abstract Writable call(RPC.RpcKind rpcKind, String protocol,
    Writable param, long receiveTime) throws Exception;
```

2. 客户端 Stub 程序

客户端的 Stub 可以看作是一个代理对象, 它会将请求程序的 RPC 调用序列化, 并调用 Client.call() 方法将这个请求发送给远程服务器, 这些实现对于客户端调用程序是完全透明的。

客户端 Stub 程序做的第一件事情就是将 RPC 请求序列化, Hadoop 2.X 默认使用 protobuf 作为序列化工具, 当然 Hadoop RPC 框架也支持其他的序列化框架。Hadoop 定义了 RpcEngine 接口抽象使用不同序列化框架的 RPC 引擎, 如图 2-2 所示, RpcEngine 接口包括两个重要的方法。

- **getProxy():** 客户端会调用 RpcEngine.getProxy() 方法获取一个本地接口的代理对象, 然后在这个代理对象上调用本地接口的方法。getProxy() 方法的实现采用了 Java 动态代理机制, 客户端调用程序在代理对象上的调用会由一个 RpcInvocationHandler (java.lang.reflect.InvocationHandler 的子类, 在 RpcEngine 的实现类中定义) 对象处理, 这个 RpcInvocationHandler 会将请求序列化 (使用 RpcEngine 实现类定义的序列化方式) 并调用 Client.call() 方法将请求发送到远程服务器。当远程服务器发回响应信息后, RpcInvocationHandler 会将响应信息反序列化并返回给调用程序, 这一切通过 Java 动态代理机制对于调用程序是完全透明的, 就像本地调用一样。
- **getServer():** 该方法用于产生一个 RPC Server 对象, 服务器会启动这个 Server 对象监听从客户端发来的请求。成功从网络接收请求数据后, Server 对象会调用 RpcInvoker (在 RpcEngine 的实现类中定义) 对象处理这个请求。

RpcEngine 目前有两个子类, 其中 WritableRpcEngine 用于描述使用 Hadoop 自带的 Writable 作为序列化工具的 RPC 引擎; ProtobufRpcEngine 用于描述使用 protobuf 作为序列化工具的 RPC 引擎。如图 2-3 所示, WritableRpcEngine 和 ProtobufRpcEngine 都定义了若干内部类, 在

这里我们简单介绍一下这些类的作用，在 Hadoop RPC 实现小节中会详细介绍这些类之间的关系以及代码实现细节。

```

1 RpcEngine
2
3 @getProxy(Class<T>, long, InetSocketAddress, UserGroupInformation, Configuration, SocketFactory, int, RetryPolicy) ProtocolProxy<T>
4 @getProxy(Class<T>, long, InetSocketAddress, UserGroupInformation, Configuration, SocketFactory, int, RetryPolicy, AtomicBoolean) tocolProxy<T>
5 @getServer(Class<?>, Object, String, int, int, int, int, boolean, Configuration, SecretManager<? extends TokenIdentifier>, String) Server
6 @getProtocolMetaInfoProxy(ConnectionId, Configuration, SocketFactory) ProtocolProxy<ProtocolMetaInfoPB>

```

图 2-2 RpcEngine 接口定义

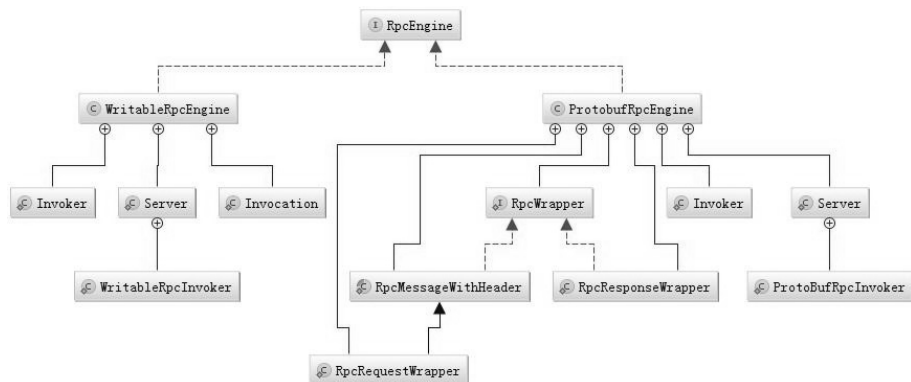


图 2-3 RpcEngine 继承结构图

(1) WritableRpcEngine

- **Invoker:** WritableRpcEngine.Invoker 是 InvocationHandler 的子类（Java 动态代理框架中的处理类）。客户端首先会调用 WritableRpcEngine.getProxy() 获取一个本地接口（例如 ClientProtocol）的代理对象，然后在这个代理对象上调用本地接口的方法。通过 Java 动态代理机制的处理，这个调用会被 WritableRpcEngine.Invoker 类的 invoke() 方法响应。WritableRpcEngine.Invoker.invoke() 方法会使用 Writable 序列化框架将 RPC 请求以及参数序列化，然后构造一个 Invocation 对象（WritableRpcEngine 的子类，实现了 Writable 接口）包装序列化的 RPC 请求以及参数，最后调用 Client.call() 方法将这个 Invocation 对象发送到远程服务器，并等待远程服务器的响应信息。成功获取了服务器发回的响应信息后，Invoker 会将响应信息反序列化并返回给客户端。
- **Invocation:** 在通信模块小节中我们介绍了 Client.call() 方法只能发送 Writable 类型的 RPC 请求，这样定义是为了让发送接口更通用，不同框架序列化的请求只需包装成一个 Writable 对象就可以通过 Client 发送。这里的 Invocation 内部类就是为了包装 Writable 的 RPC 请求，它保存了客户端在什么接口上调用什么方法，以及这个方法的参数等调用信息。Invoker 构造了这个对象之后，就会调用 Client.call() 方法将这个对象发送到远程服务器。
- **Server:** WritableRpcEngine.Server 是 org.apache.hadoop.ipc.Server 类的子类，服务器代码会调用 WritableRpcEngine.getServer() 方法获取一个 WritableRpcEngine.Server 对象，WritableRpcEngine.Server 类继承了的 Server 大部分方法，它会在 Socket 上监听

RPC 请求，并调用 `WritableRpcInvoker` 类的 `call()` 方法响应这个请求。

- **WritableRpcInvoker:** `WritableRpcInvoker` 是 `WritableRpcEngine.Server` 的内部类，它实现了 `RpcInvoker` 接口，用于响应远程客户端的请求。`WritableRpcInvoker` 会先使用 `Writable` 序列化工具反序列化请求信息以及请求参数，然后根据请求信息反射调用服务程序，并将响应结果包装成一个 `Writable` 对象返回。

(2) ProtobufRpcEngine

- **Invoker:** 与 `WritableRpcEngine.Invoker` 类相同，都是用于处理客户端发送 RPC 请求的。不同处有 4 点：①RPC 请求信息以及请求参数的序列化使用了 `protobuf` 工具；②使用 `RPCRequestWrapper` 类包装 RPC 请求（包括请求信息以及请求参数）；③使用 `RPCResponseWrapper` 类包装服务器发回的 RPC 响应信息；④使用 `protobuf` 反序列化远程服务器发回的 RPC 响应信息。
- **RpcRequestWrapper:** 我们知道 `Client.call()` 方法只接受 `Writable` 类型的 RPC 请求，而 `ProtobufRpcEngine.Invoker` 是使用 `protobuf` 序列化 RPC 请求的，这里就需要一个包装类 `RpcRequestWrapper` 将 `protobuf` 格式的请求包装成 `Writable` 类型。`RpcRequestWrapper` 保存了两部分信息，这两部分信息都是使用 `protobuf` 序列化的——①RPC 请求头：RPC 请求头记录了客户端在什么接口上调用了什么方法；②请求参数：方法调用的参数。`Invoker` 类构造了 `RpcRequestWrapper` 对象后，会调用 `Client.call()` 方法将这个请求发送到远程服务器。
- **RpcResponseWrapper:** `Client.call()` 方法返回的响应信息也是 `Writable` 类型的，但是服务器端使用了 `protobuf` 序列化 RPC 响应信息，这就需要定义一个包装类 `RpcResponseWrapper` 将这个 `protobuf` 序列化的响应信息包装成 `Writable` 类型。
- **Server:** 与 `WritableRpcEngine.Server` 类的功能类似，它会在 `Socket` 上监听 RPC 请求，并调用 `ProtoBufRpcInvoker` 类的 `call()` 方法响应这个请求。
- **ProtoBufRpcInvoker:** 与 `WritableRpcInvoker` 类的功能类似，用于响应远程客户端的请求。不同之处有如下几点：①使用了 `protobuf` 这个序列化工具；②使用了 `RPCRequestWrapper` 类包装 RPC 请求（包括请求信息以及请求参数）；③使用了 `RPCResponseWrapper` 类包装服务器发回的 RPC 响应信息。

了解了客户端 `Stub` 程序的功能，我们再来学习客户端调用程序获得 `Stub` 引用的流程。客户端会调用 `RPC.getProtocolProxy()` 方法获取某个本地接口（例如 `ClientProtocol`）的代理对象，之后调用程序就可以在该代理对象上调用本地接口的方法了。如下代码所示，`RPC.getProtocolProxy()` 方法会首先调用 `getProtocolEngine()` 获取当前 RPC 类的序列化引擎（可能是 `WritableRpcEngine` 或者 `ProtobufRpcEngine`），然后调用 `RpcEngine.getProxy()` 方法获取代理对象。`RpcEngine.getProxy()` 方法的具体实现我们将在客户端获取 `Proxy` 对象小节中介绍。

```
public static <T> ProtocolProxy<T> getProtocolProxy(Class<T> protocol,
                                                    long clientVersion,
                                                    InetAddress addr,
                                                    UserGroupInformation ticket,
```

```

        Configuration conf,
        SocketFactory factory,
        int rpcTimeout,
        RetryPolicy connectionRetryPolicy,
        AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    return getProtocolEngine(protocol, conf).getProxy(protocol, clientVersion,
        addr, ticket, conf, factory, rpcTimeout, connectionRetryPolicy,
        fallbackToSimpleAuth);
}

```

至此，与客户端 Stub 相关的内容就介绍完了。图 2-4 给出了调用程序通过 `RPC.getProxy()` 获取一个 `TestProtocol` 接口的代理对象，然后在代理对象上调用 `TestProtocol.test()` 方法的流程。这个流程的实现细节将会在客户端获取 Proxy 对象小节中介绍。

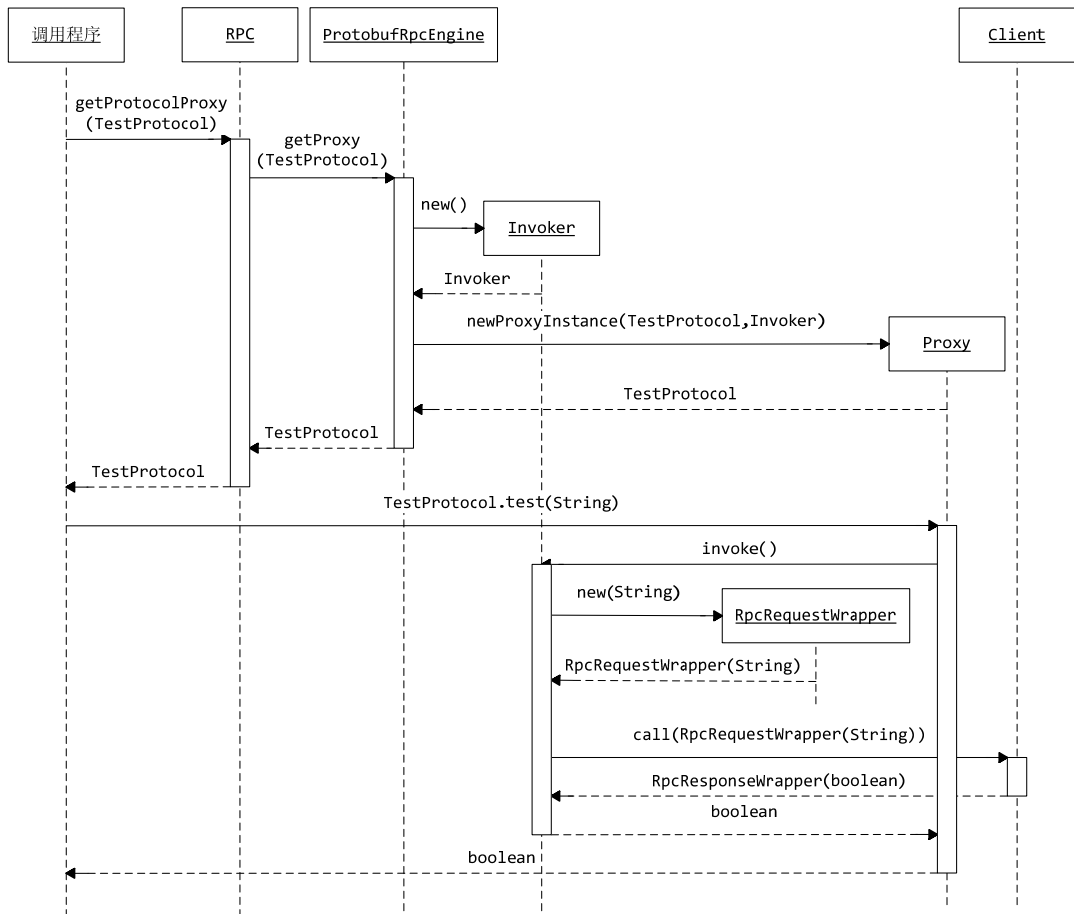


图 2-4 客户端调用流程

3. 服务器端 Stub 程序

服务器端 Stub 程序会将通信模块接收的数据反序列化，然后调用服务程序对应的方法响应这个 RPC 请求。在通信模块小节中我们介绍了 Server 类是服务器端从网络接收 RPC 请求的类，成功地接收一个 RPC 请求后，Server 会调用 call()方法响应这个请求。

RPC.Server.call()实现了父类 Server 的抽象方法 call()。RPC.Server.call()的代码如下所示，它首先调用 getRpcInvoker()获取一个 RpcInvoker 对象，然后调用 RpcInvoker.call()方法响应这个 RPC 请求，这个 RpcInvoker 对象可以理解为服务器端 Stub 程序的实现。

```
public Writable call(RPC.RpcKind rpcKind, String protocol,
    Writable rpcRequest, long receiveTime) throws Exception {
    return getRpcInvoker(rpcKind).call(this, protocol, rpcRequest,
        receiveTime);
}
```

如图 2-3 所示，不同的 RpcEngine 会实现自己的 Server 对象 (RPC.Server 的子类)，Server 对象又会实现一个内部的 RpcInvoker 对象，例如 ProtobufRpcEngine 实现了内部类 Server，而 Server 类又定义了自己的 ProtoBufRpcInvoker 类。这个 ProtoBufRpcInvoker 对象会使用 protobuf 反序列化 RPC 请求 (RpcRequestWrapper 类型)，然后调用服务程序响应这个请求，最后 ProtoBufRpcInvoker 会将响应消息序列化包装成一个 RpcResponseWrapper 对象并返回。整个服务器端 Stub 处理 RPC 请求的过程如图 2-5 所示。

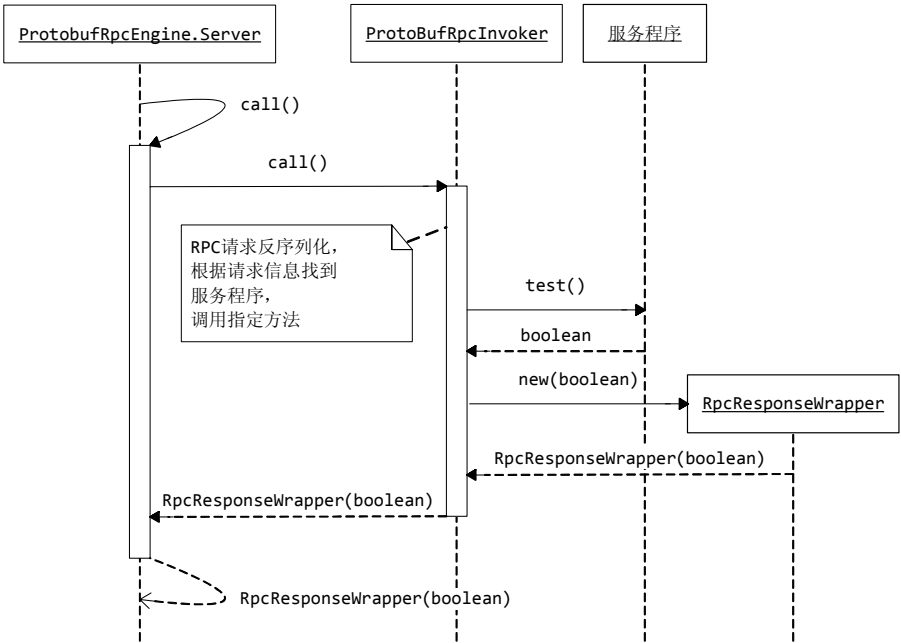


图 2-5 服务器端响应流程

那么，服务器端如何获取这个 Server 对象呢？服务器端会调用 `RPC.Builder.build()` 方法获取 Server 对象的引用，`build()` 方法是一个工厂方法，它首先检查工厂的参数是否设置了，然后会调用 `getProtocolEngine()` 方法获取当前 RPC 类配置的 `RpcEngine`，最后调用 `RpcEngine.getServer()` 获取使用特定序列化框架的 Server 类，例如 `ProtobufRpcEngine.getServer()` 会返回 `ProtobufRpcEngine.Server` 类，`ProtobufRpcEngine.Server` 就会使用 `ProtoBufRpcInvoker` 作为服务器的 Stub 代理程序。`RpcEngine.getServer()` 方法的实现请参考服务器获取 Server 对象小节。

```
public Server build() throws IOException, HadoopIllegalArgumentException {
    // 检查参数
    if (this.conf == null) {
        throw new HadoopIllegalArgumentException("conf is not set");
    }
    if (this.protocol == null) {
        throw new HadoopIllegalArgumentException("protocol is not set");
    }
    if (this.instance == null) {
        throw new HadoopIllegalArgumentException("instance is not set");
    }

    // 调用 RpcEngine.getServer() 获取 Server 对象
    return getProtocolEngine(this.protocol, this.conf).getServer(
        this.protocol, this.instance, this.bindAddress, this.port,
        this.numHandlers, this.numReaders, this.queueSizePerHandler,
        this.verbose, this.conf, this.secretManager, this.portRangeConfig);
}
```

2.2 Hadoop RPC 的使用

上一节我们介绍了 Hadoop RPC 框架结构，本节则重点介绍如何使用 Hadoop RPC，同时会以 `ClientProtocol.rename()` 调用为例介绍整个调用流程的代码实现。

2.2.1 Hadoop RPC 使用概述

图 2-6 给出了 `DFSClient` 调用 `ClientProtocol.rename()` 方法的流程图。我们首先看一下 RPC 协议的定义部分。`ClientProtocol` 协议定义了 HDFS 客户端与名字节点交互的所有方法，但是 `ClientProtocol` 协议中方法的参数是无法在网络中传输的，需要对参数进行序列化操作，所以 HDFS 又定义了 `ClientNamenodeProtocolPB` 协议。`ClientNamenodeProtocolPB` 协议包含了 `ClientProtocol` 定义的所有方法，但是参数却是使用 `protobuf` 序列化后的格式。这里还是以 `rename()` 方法为例，`ClientNamenodeProtocolPB` 将 `ClientProtocol` 中 `rename(String, String)` 方法的两个参数抽象成一个 `RenameRequestProto` 对象，`rename()` 方法的签名也就变成了 `rename(RenameRequestProto)`。这里的 `RenameRequestProto` 对象是通过 `protobuf` 序列化后的对

象，是可以在网络上传输的对象。

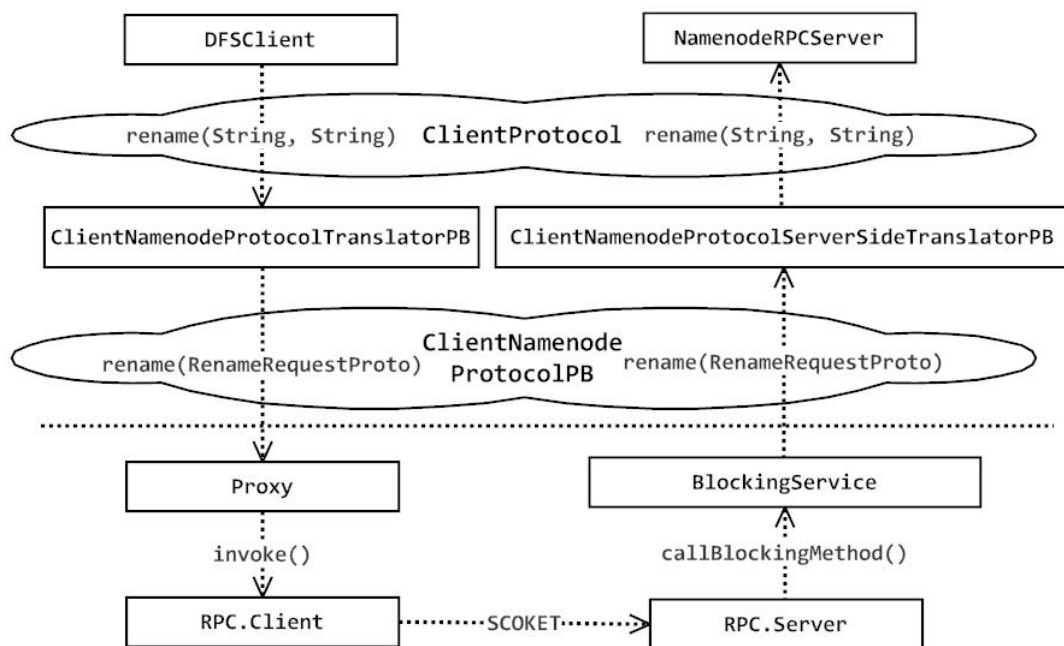


图 2-6 ClientProtocol.rename()调用流程图

了解了这两个接口之后，我们再来看 rename()调用就比较简单了。为了将不可以序列化的 ClientProtocol 接口调用转换为可以序列化的 ClientNamenodeProtocolPB 接口调用，HDFS 引入了两个适配器类（使用适配器模式的类）进行接口适配。

- ClientNamenodeProtocolTranslatorPB 类：作为 Client 侧的适配器类，实现了 ClientProtocol 接口，它内部拥有一个实现了 ClientNamenodeProtocolPB 接口的对象，可以将 ClientProtocol 调用适配成 ClientNamenodeProtocolPB 调用。以 rename()调用为例，ClientNamenodeProtocolTranslatorPB 将 rename(String, String)调用中的两个 String 参数序列化成一个 RenameRequestProto 对象，然后调用 ClientNamenodeProtocolPB 对象的 rename(RenameRequestProto)方法，这样就完成了 ClientProtocol 接口到 ClientNamenodeProtocolPB 接口的适配。
- ClientNamenodeProtocolServerSideTranslatorPB 类：作为 Server 侧的适配器类，实现了 ClientNamenodeProtocolPB 接口，同时内部拥有一个实现了 ClientProtocol 接口的对象（NameNodeRpcServer），可以将 ClientNamenodeProtocolPB 调用适配成 ClientProtocol 调用。还是以 rename()调用为例，ClientNamenodeProtocolServerSideTranslatorPB 将 rename(RenameRequestProto)调用的 RenameRequestProto 对象反序列化成两个 String 对象，之后调用 NameNodeRpcServer（实现了 ClientProtocol）类的 rename(String, String)方法执行重命名操作。

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

客户端调用 `ClientNamenodeProtocolTranslatorPB.rename(String,String)` 方法后, `ClientNamenodeProtocolTranslatorPB` 对象是如何将 `rename` 请求发送到 Server 端的呢? 如图 2-7 所示, `ClientNamenodeProtocolTranslatorPB` 类持有一个实现了 `ClientNamenodeProtocolPB` 接口的对象, `ClientNamenodeProtocolTranslatorPB` 会将 `ClientProtocol.rename(String, String)` 调用的参数序列化成一个 `RenameRequestProto` 对象, 然后在 `ClientNamenodeProtocolPB` 对象上调用 `rename(RenameRequestProto)` 方法。Hadoop RPC 在这里非常巧妙地使用了 Java 动态代理机制, `ClientNamenodeProtocolTranslatorPB` 持有的 `ClientNamenodeProtocolPB` 对象其实是通过 Java 动态代理机制获取的一个 `ClientNamenodeProtocolPB` 接口的代理对象 (调用 `RPC.getProtocolProxy()` 方法获取), 这个代理对象内部封装了一个 `ProtobufRpcEngine.Invoker` 对象。对 `ClientNamenodeProtocolPB` 接口的调用都会由这个 `Invoker` 对象的 `invoke()` 方法代理, `Invoker.invoke()` 方法会首先构造一个描述 RPC 调用信息的对象 `RequestHeaderProto` (使用 `protobuf` 序列化的), 记录了客户端在什么协议上调用了什么方法 (在 `ClientProtocol` 协议上调用了 `rename` 方法), 然后将 `RequestHeaderProto` 对象以及调用参数对象 `RenameRequestProto` 包装成一个 `RpcRequestWrapper` 对象, 最后就可以调用底层 `RPC.Client` 类提供的 `call()` 方法将 `rename` 请求发送到远程服务器了。由于这里巧妙地使用了动态代理机制, `Invoker` 调用 `RPC.Client.call()` 方法发送请求到远端的所有操作对 `ClientNamenodeProtocolTranslatorPB` 对象都是透明的, 客户端调用 `ClientNamenodeProtocolTranslatorPB.rename()` 方法就与调用本地方法一样。

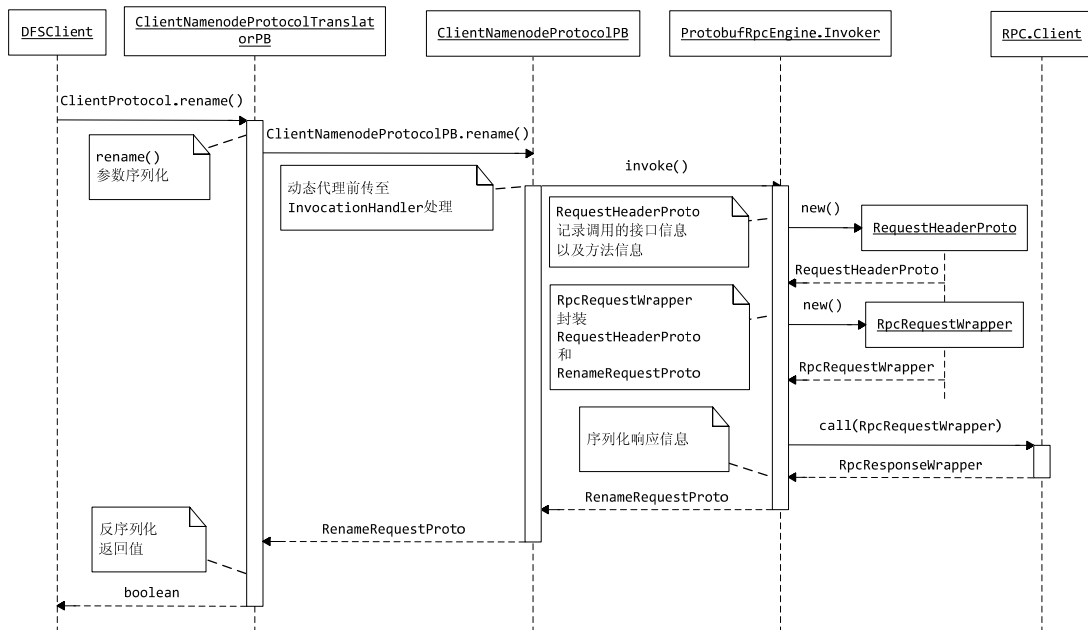


图 2-7 客户端发送 `rename` 请求流程

那么，当 `rename` 请求到达服务器端之后，服务器端是如何响应的呢？如图 2-8 所示，`NameNodeRpcServer` 会启动一个 `RPC Server` 监听来自客户端的所有 `RPC` 请求，当 `RPC Server` 在网络上监听到一个 `RPC` 请求时，它会从网络中解析这个请求，然后构造一个 `ProtoBufRpcInvoker` 对象来处理这个请求。`ProtoBufRpcInvoker` 对象会将请求数据反序列化，解析出调用信息和调用参数。之后 `ProtoBufRpcInvoker` 对象会根据调用信息中的调用接口信息（在这个例子中是 `ClientProtocol`）查找实现了 `ClientProtocol` 接口的 `BlockingService` 对象，这个 `BlockingService` 对象持有一个实现了 `ClientNamenodeProtocolPB` 接口的 `ClientNamenodeProtocolServerSideTranslatorPB` 对象，用于响应 `ClientNamenodeProtocolPB` 接口上的调用。获取了 `BlockingService` 对象后，`ProtoBufRpcInvoker` 利用调用信息中的调用方法信息（在这个例子中是 `rename`）和调用参数对象（`RenameRequestProto`）调用 `BlockingService.callBlockingMethod()` 方法响应 `RPC` 请求。`callBlockingMethod()` 方法根据调用方法信息判断这是一个 `ClientNamenodeProtocolPB.rename()` 调用，它会在 `ClientNamenodeProtocolServerSideTranslatorPB` 对象上调用 `rename(RenameRequestProto)` 方法响应，`ClientNamenodeProtocolServerSideTranslatorPB.rename()` 方法会将 `RenameRequestProto` 参数反序列化成两个 `String` 参数，然后在自己持有的实现了 `ClientProtocol` 接口的 `NameNodeRpcServer` 上调用 `rename(String,String)` 方法响应，`NameNodeRpcServer.rename()` 方法会在 `Namenode` 的命名空间中更改指定 `HDFS` 文件的名称，最后返回响应信息。`ClientNamenodeProtocolServerSideTranslatorPB` 接收到 `NameNodeRpcServer` 的响应信息后会将这个响应包装成一个 `protobuf` 序列化的 `RenameResponseProto` 对象，然后返回到 `ProtoBufRpcInvoker` 对象。`ProtoBufRpcInvoker` 接收到 `RenameResponseProto` 这个响应对象后，由于 `ProtoBufRpcInvoker.call()` 方法的返回值定义是 `Writable` 类型的，所以 `ProtoBufRpcInvoker` 会构造一个 `RpcResponseWrapper` 对象包装 `RenameResponseProto`，然后将这个对象返回给 `RPC Client`。

学习了 `ClientProtocol.rename()` 调用的流程后，我们可以将 Hadoop `RPC` 框架的使用抽象为如下几个步骤。

- 定义 `RPC` 协议：`RPC` 协议是客户端和服务端之间 `RPC` 调用的接口，只有定义了 `RPC` 协议，客户端才知道服务器对外提供了哪些服务。这里以 `ClientProtocol` 为例，`ClientProtocol` 定义了 `Namenode` 服务器与 `HDFS` 客户端之间的接口，`HDFS` 客户端调用 `ClientProtocol.rename()` 方法，`Namenode` 服务器就会更改指定 `HDFS` 文件的文件名。
- 实现 `RPC` 协议：服务器端的服务程序需要实现 `RPC` 协议，当 `RPC` 调用通过网络到达服务器时，实现了 `RPC` 协议的服务程序会响应这个 `RPC` 调用。还是以 `ClientProtocol` 为例，`Namenode` 端的 `NameNodeRpcServer` 类实现了 `ClientProtocol` 协议，当 `ClientProtocol` `RPC` 请求到达 `Namenode` 时，会由 `NameNodeRpcServer` 类响应这个 `RPC` 请求。
- 客户端获取代理对象：客户端需要调用 `RPC.getProtocolProxy()` 方法获取一个 `RPC` 协议的代理对象，之后客户端调用程序就可以在这个代理对象上调用 `RPC` 协议的方法，通过 `Java` 动态代理机制，这个 `RPC` 请求会由一个 `InvocationHandler` 代理对象处理，`InvocationHandler` 对象会将 `RPC` 调用信息和调用参数序列化，最后通过调用 `Client.call()` 方法将这个请求发送到远程服务器。

- 服务器构造并启动 RPC Server: 服务器需要调用 `RPC.Builder.build()` 方法构造一个 `Server` 对象, 然后调用 `Server.start()` 方法启动这个 `Server` 对象响应来自客户端的 RPC 请求。例如对于 `NameNodeRpcServer` 类, 它会构造两个 `Server` 对象分别响应来自 HDFS 客户端和数据节点的 RPC 请求。

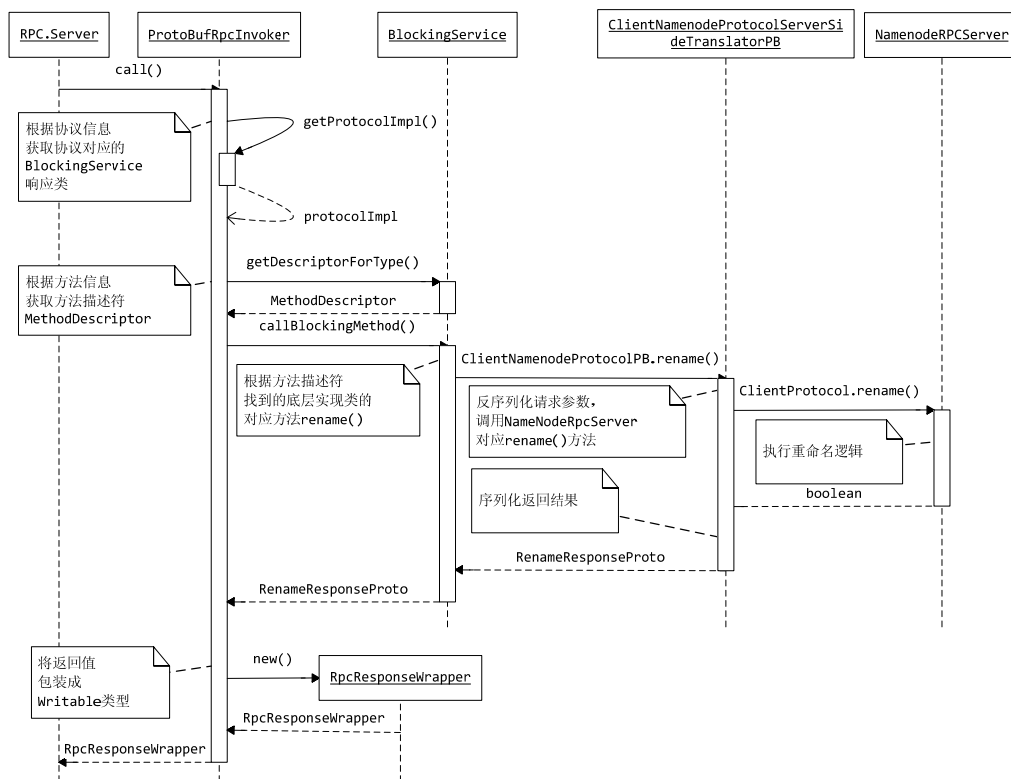


图 2-8 客户端响应 rename 请求流程

2.2.2 定义 RPC 协议

在第 1 章的 HDFS 概述小节中我们介绍了 HDFS 为各个节点之间抽象了不同的接口, 也就是 RPC 协议, 例如 `ClientProtocol` 抽象了客户端与名字节点之间的通信协议, 而 `DatanodeProtocol` 则抽象了数据节点与名字节点之间的通信协议。通过上一节的介绍, 我们知道定义 RPC 协议是使用 Hadoop RPC 框架的第一步, 而对于 HDFS 中的所有 RPC 协议, 都会存在一个如图 2-6 所示的协议栈。例如对于 `DatanodeProtocol`, 存在对应的支持序列化的接口 `DatanodeProtocolPB`, 同时还存在 `DatanodeProtocolClientSideTranslatorPB` 以及 `DatanodeProtocolServerSideTranslatorPB` 两个类用于进行协议适配, 读者对比理解即可。本节还是以 `ClientProtocol` 作为例子来介绍这些支持类的代码实现。

1. ClientProtocol 协议

ClientProtocol 协议定义了 HDFS 客户端与名字节点交互的所有接口方法（例如 rename()、mkdir()等），ClientProtocol 定义的具体方法我们在 HDFS 概述小节中已经介绍过了，本节主要介绍 ClientProtocol 的子类。ClientProtocol 的类结构如图 2-9 所示，定义了两个子类。

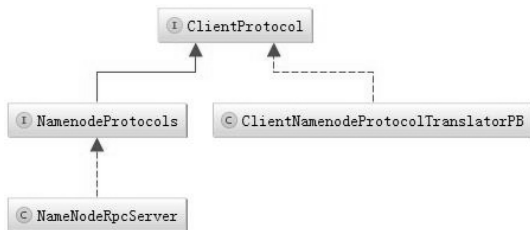


图 2-9 ClientProtocol 继承结构图

- ClientNameNodeProtocolTranslatorPB——这个类是 RPC 客户端侧最重要的类之一，它将客户端的请求参数封装成可以序列化的 protobuf 格式，然后通过代理类（实现 ClientNameNodeProtocolPB 接口）发送出去。
- NameNodeRpcServer——NameNode 侧响应 ClientProtocol 调用的类，它会执行 HDFS 操作并将操作结果返回。

2. ClientNameNodeProtocolPB 协议

我们知道 ClientProtocol 是 HDFS 客户端与名字节点之间的接口，但是在 RPC 调用中，RPC 方法的参数和返回值都必须是可序列化的，所以 HDFS 定义了一个新的 ClientNameNodeProtocolPB 协议对 ClientProtocol 协议中方法的参数和返回值进行包装。ClientNameNodeProtocolPB 协议定义的方法与 ClientProtocol 一样，但是参数和返回值却是使用 protobuf 序列化后的格式，这样 ClientNameNodeProtocolPB 请求就可以通过网络传输。为了便于理解，大家可以把 ClientNameNodeProtocolPB 理解为使用 protobuf 定义参数和返回值的 ClientProtocol 协议。

那么如何定义 ClientNameNodeProtocolPB 协议呢？Hadoop RPC 在这里使用了 protobuf 工具。使用 protobuf 工具的第一步是定义一个 ClientNameNodeProtocol.proto 文件，这个文件中包含了 RPC 协议 ClientNameNodeProtocol 的定义，ClientNameNodeProtocol 是 protobuf 产生的支持 ClientProtocol RPC 调用的协议接口类，ClientNameNodeProtocolPB 是 ClientNameNodeProtocol.BlockingInterface 接口的子类。ClientNameNodeProtocol.proto 文件中 ClientNameNodeProtocol 协议的定义如下代码所示，可以看到 ClientNameNodeProtocol 就是将 ClientProtocol 的参数全部抽象为一个 *RequestProto 对象，而将返回值抽象为一个 *ResponseProto 对象。这里还是以 rename()方法为例，ClientNameNodeProtocol.rename()方法的参数是 RenameRequestProto，而返回值是 RenameResponseProto。RenameRequestProto 和 RenameResponseProto 是 protobuf 定义的用于封装参数以及返回值的类，它们的结构也是在 ClientNameNodeProtocol.proto 文件中定义的。

```
service ClientNamenodeProtocol {
  rpc getBlockLocations(GetBlockLocationsRequestProto)
    returns(GetBlockLocationsResponseProto);
  rpc getServerDefaults(GetServerDefaultsRequestProto)
    returns(GetServerDefaultsResponseProto);
  rpc create(CreateRequestProto) returns (CreateResponseProto);
  rpc append(AppendRequestProto) returns (AppendResponseProto);
  rpc setReplication(SetReplicationRequestProto)
    returns(SetReplicationResponseProto);
  rpc setPermission(SetPermissionRequestProto)
    returns(SetPermissionResponseProto);
  // ...
}
```

`ClientNamenodeProtocol.proto` 文件中定义的第二部分是调用过程中的请求参数和响应参数的声明，也就是上面描述的 `RenameRequestProto` 和 `RenameResponseProto` 的声明（注意这两个类是写好声明之后，执行 `protoc.exe` 动态生成的），`RenameRequestProto` 和 `RenameResponseProto` 对象都是可以通过 `protobuf` 工具序列化的。如下代码所示，`RenameRequestProto` 的定义中包含了 `ClientProtocol.rename()` 方法的两个 `String` 类型的参数（`src&dst`），`RenameResponseProto` 则包含了 `ClientProtocol.rename()` 方法的 `bool` 类型的返回值（重命名操作的执行结果）。

```
message RenameRequestProto {
  required string src = 1;  // rename() 方法的第一个参数 src
  required string dst = 2;  // rename() 方法的第二个参数 dst
}

message RenameResponseProto {
  required bool result = 1;  // 封装返回结果，布尔类型
}
```

在 `ClientNamenodeProtocol.proto` 文件中定义了 `ClientNamenodeProtocol` 后，就可以使用 `protobuf` 工具生成 `ClientNamenodeProtocol` 类了，生成的 `ClientNamenodeProtocol.BlockingInterface` 接口的代码如下所示。我们注意到 `BlockingInterface` 接口中定义的所有方法都只有两个参数。

- **controller 参数：**这个参数没有用到，方法调用时设置为 `Null`。
- **request 参数：**封装原有方法调用的参数，例如 `ClientProtocol.rename(String, String)` 方法有两个 `String` 类型的参数（`src & dst`），在 `ClientNamenodeProtocol.BlockingInterface` 接口中这两个参数是用 `RenameRequestProto` 封装的，`RenameRequestProto` 是可以通过 `protobuf` 序列化的类，结构是在 `ClientNamenodeProtocol.proto` 文件中定义的。

```
public static abstract class ClientNamenodeProtocol
  implements com.google.protobuf.Service {
  protected ClientNamenodeProtocol() {}
  // ...
  public interface BlockingInterface {
    public RenameResponseProto rename(
```



```

        com.google.protobuf.RpcController controller,
        RenameRequestProto request)
        throws com.google.protobuf.ServiceException;

    public CompleteResponseProto complete(
        com.google.protobuf.RpcController controller,
        CompleteRequestProto request)
        throws com.google.protobuf.ServiceException;

    public ReportBadBlocksResponseProto reportBadBlocks(
        com.google.protobuf.RpcController controller,
        ReportBadBlocksRequestProto request)
        throws com.google.protobuf.ServiceException;

    public ConcatResponseProto concat(
        com.google.protobuf.RpcController controller,
        ConcatRequestProto request)
        throws com.google.protobuf.ServiceException;
    // ...
}
}

```

ClientNamenodeProtocolPB 接口是 ClientNamenodeProtocol.BlockingInterface 接口的子类，它完全继承了 ClientNamenodeProtocol.BlockingInterface 接口中方法的定义。读者可能会问，这里直接使用 ClientNamenodeProtocol 作为序列化协议就可以啊，为什么还要再定义一个 ClientNamenodeProtocolPB 协议呢？之所以使用 ClientNamenodeProtocolPB 作为序列化协议类是为了添加 Annotation 支持，因为 ClientNamenodeProtocol 的代码是 protobuf 自动生成的，不可以添加 Annotation 支持。ClientNamenodeProtocolPB 接口的定义如下：

```

@InterfaceAudience.Private
@InterfaceStability.Stable
@KerberosInfo(
    serverPrincipal = DFSConfigKeys.DFS_NAMENODE_KERBEROS_PRINCIPAL_KEY)
@TokenInfo(DelegationTokenSelector.class)
@ProtocolInfo(protocolName = HdfsConstants.CLIENT_NAMENODE_PROTOCOL_NAME,
    protocolVersion = 1)
// ClientNamenodeProtocolPB 是 ClientNamenodeProtocol.BlockingInterface 的子类
// ClientNamenodeProtocol 则是由 protobuf 工具生成的 RPC 调用接口类
public interface ClientNamenodeProtocolPB extends
    ClientNamenodeProtocol.BlockingInterface {
}

```

3. ClientNamenodeProtocolTranslatorPB 类

如图 2-6 所示，DFSClient 会持有一个 ClientNamenodeProtocolTranslatorPB 类的引用，用于将 ClientProtocol 协议上的请求适配成 ClientNamenodeProtocolPB 协议的请求。如下代码所示，ClientNamenodeProtocolTranslatorPB 类实现了 ClientProtocol 接口，并持有一个实现了

ClientNamenodeProtocolPB 接口的 rpcProxy 对象。

```
// ClientNamenodeProtocolTranslatorPB 实现了 ClientProtocol 接口
public class ClientNamenodeProtocolTranslatorPB implements
    ProtocolMetaInterface, ClientProtocol, Closeable, ProtocolTranslator {
    // 持有一个实现 ClientNamenodeProtocolPB 接口的 rpcProxy 对象
    final private ClientNamenodeProtocolPB rpcProxy;
    // ...
}
```

ClientNamenodeProtocolTranslatorPB 会将 ClientProtocol 请求的参数序列化，然后调用 rpcProxy 对象（实现了 ClientNamenodeProtocolPB 接口）上的对应方法，这样就完成了适配操作。我们还是以 rename() 方法调用作为例子，如下代码所示，ClientNamenodeProtocolTranslatorPB 执行了两个操作来完成 ClientProtocol.rename() 调用的适配功能。

- 将 ClientProtocol.rename() 调用的两个 String 类型的参数封装成 RenameRequestProto 对象。
- 调用 ClientNamenodeProtocolPB.rename() 方法，然后反序列化调用结果并返回。

```
public boolean rename(String src, String dst) throws UnresolvedLinkException,
    IOException {
    // 构建 pb 参数
    RenameRequestProto req = RenameRequestProto.newBuilder()
        .setSrc(src)
        .setDst(dst).build();
    try {
        // 调用底层 impl 对应方法，返回结果
        return rpcProxy.rename(null, req).getResult();
    } catch (ServiceException e) {
        throw ProtobufHelper.getRemoteException(e);
    }
}
```

至于这里的 rpcProxy 对象（实现了 ClientNamenodeProtocolPB 接口），则是使用 Java 动态代理机制获取的 ClientNamenodeProtocolPB 接口的代理对象。在这个对象上的调用会由 ProtobufRpcEngine.Invoker 对象代理，这个 Invoker 对象的 invoke() 方法会调用底层的 RPC.Client 类提供的 call() 方法将 ClientNamenodeProtocolPB.rename() 请求发送到远程服务器，并等待远程服务器返回响应信息。获取 ClientNamenodeProtocolPB 接口代理对象的流程我们将在客户端获取 Proxy 对象小节中详细介绍。

4. ClientNamenodeProtocolServerSideTranslatorPB 类

ClientNamenodeProtocolTranslatorPB 类是 RPC Client 侧的适配器类，ClientNamenodeProtocolServerSideTranslatorPB 则是 RPC Server 侧的适配器类。Client 侧的适配器用于将 ClientProtocol 的请求转换为 ClientNamenodeProtocolPB 请求，那么 Server 侧的适配器则是将 ClientNamenodeProtocolPB 请求转换为 ClientProtocol 请求。如下代码所示，ClientNamenodeProtocolServerSideTranslatorPB 实现了 ClientNamenodeProtocolPB 接口，底层持有一个实现了

ClientProtocol 接口的 server 对象。

```
public class ClientNameNodeProtocolServerSideTranslatorPB implements
    ClientNameNodeProtocolPB {
    // 底层实现类
    final private ClientProtocol server;
    // ...
}
```

ClientNameNodeProtocolServerSideTranslatorPB 只需将 ClientNameNodeProtocolPB 请求的参数反序列化, 然后在 ClientProtocol 服务对象 server 上调用对应的方法即可完成适配工作。这里还是以 rename() 方法为例, ClientNameNodeProtocolServerSideTranslatorPB.rename() 会首先反序列化请求参数 RenameRequestProto 对象, 然后调用 server 对象的 rename() 方法执行重命名操作, 最后将返回值序列化为 RenameResponseProto 对象并返回。

```
public RenameResponseProto rename(RpcController controller,
    RenameRequestProto req) throws ServiceException {
    try {
        boolean result = server.rename(req.getSrc(), req.getDst());
        return RenameResponseProto.newBuilder().setResult(result).build();
    } catch (IOException e) {
        throw new ServiceException(e);
    }
}
```

ClientNameNodeProtocolPB 持有的 server 对象实际上是 NameNodeRpcServer 的引用, 我们知道 NameNodeRpcServer 是 ClientProtocol 的子类, 它会响应 ClientProtocol 的请求, 修改名字节点的命名空间, 执行对应的 HDFS 逻辑。

2.2.3 客户端获取 Proxy 对象

我们知道 DFSClient 会持有一个 ClientProtocol 对象向 Namenode 发送请求, 在 Hadoop RPC 框架概述小节中我们介绍了 DFSClient 持有的 ClientProtocol 对象其实是 ClientNameNodeProtocolTranslatorPB 的实例 (ClientNameNodeProtocolTranslatorPB 实现了 ClientProtocol 接口)。在上一节中我们介绍了 ClientNameNodeProtocolTranslatorPB 对象会持有一个实现了 ClientNameNodeProtocolPB 接口的对象, 这个对象是通过 Java 动态代理机制获取的 ClientNameNodeProtocolPB 接口的代理对象。

接下来 ClientNameNodeProtocolTranslatorPB 对象会将 ClientProtocol 调用的参数使用 protobuf 序列化, 然后调用 ClientNameNodeProtocolPB 接口代理类的对应方法将这个请求发送出去。在 ClientNameNodeProtocolPB 代理对象上的调用都会由一个 ProtobufRpcEngine.Invoker 对象代理, Invoker 对象会构造请求头 (在什么接口上调用什么方法), 然后调用 RPC.Client.call() 方法将请求头以及序列化之后的参数发送到服务器。本节就重点介绍 DFSClient 获取 ClientNameNodeProtocolTranslatorPB 对象, 以及 ClientNameNodeProtocolTranslatorPB 对象获取 ClientNameNodeProtocolPB 代理对象的流程。

我们先从一个简单的 ClientRPC 请求 (DFSClient.rename()) 开始分析, 如下代码所示, DFSClient 会在 namenode 这个引用上调用 rename2()方法。

```
public void rename(String src, String dst, Options.Rename... options)
    throws IOException {
    checkOpen();
    try {
        namenode.rename2(src, dst, options);
    } catch (RemoteException re) {
        // ...
    }
}
```

这里的 namenode 字段保存了一个实现了 ClientProtocol 接口的对象, DFSClient 通过 ProxyInfo 类来获取这个对象的引用, 而这个 ProxyInfo 对象则是通过调用 NameNodeProxies.createProxy()方法产生的。

```
final ClientProtocol namenode;
// 构造方法
this.namenode = proxyInfo.getProxy();
// 获取 proxyInfo 引用
proxyInfo = NameNodeProxies.createProxy(conf, nameNodeUri, ClientProtocol.class);
```

我们知道 ClientProtocol 对象定义了客户端与名字节点之间的所有接口, 而 Hadoop 2.X 引入了 Namenode 的 HA 机制, 也就是说, HDFS 集群中会存在两个 Namenode 实例, 同一时间 DFSClient 只会将 ClientProtocol RPC 请求发送给集群中的 Active Namenode。而当集群发生错误切换时, DFSClient 又会将请求发送给新的 Active Namenode, 这些实现对于 DFSClient 来说是透明的, DFSClient 并不知道 ClientProtocol RPC 请求发送到了哪个 Namenode, 它只需在 ClientProtocol 对象上发起 RPC 调用即可。NameNodeProxies.createProxy()方法就是用于创建支持 HA 机制的 ClientProtocol 代理对象的, 它会根据配置文件判断当前 HDFS 集群是否处于 HA 模式。对于处于 HA 模式的情况, createProxy()方法会调用 createFailoverProxyProvider()方法创建支持 HA 机制的 ClientProtocol 对象; 而对于非 HA 模式的情况, createProxy()方法则会调用 createNonHAProxy()方法创建普通的 ClientProtocol 对象。NameNodeProxies.createProxy()方法的代码如下:

```
public static <T> ProxyAndInfo<T> createProxy(Configuration conf,
    URI nameNodeUri, Class<T> xface) throws IOException {
    Class<FailoverProxyProvider<T>> failoverProxyProviderClass =
        getFailoverProxyProviderClass(conf, nameNodeUri, xface);

    if (failoverProxyProviderClass == null) {
        // 非 HA 情况
        return createNonHAProxy(conf, NameNode.getAddress(nameNodeUri), xface,
            UserGroupInformation.getCurrentUser(), true);
    } else {
        // HA 情况
        FailoverProxyProvider<T> failoverProxyProvider = NameNodeProxies
```

```

        .createFailoverProxyProvider(conf, failoverProxyProviderClass, xface,
            nameNodeUri);
    Conf config = new Conf(conf);
    T proxy = (T) RetryProxy.create(xface, failoverProxyProvider, RetryPolicies
        .failoverOnNetworkException(RetryPolicies.TRY_ONCE_THEN_FAIL,
            config.maxFailoverAttempts, config.failoverSleepBaseMillis,
            config.failoverSleepMaxMillis));

    Text dtService = HAUtil.buildTokenServiceForLogicalUri(nameNodeUri);
    return new ProxyAndInfo<T>(proxy, dtService);
}
}

```

下面我们就分析 HA 以及非 HA 情况时的 ClientProtocol 对象的构造流程。

1. 非 HA 模式

非 HA 模式的入口方法是 NameNodeProxies.createNonHAProxy(), 它会对 xface 参数也就是 RPC 接口进行判断, 然后构造并返回实现了这个接口的对象。对于获取 ClientProtocol 代理的情况, createNonHAProxy 会调用 createNNProxyWithClientProtocol() 方法创建实现了 ClientProtocol 接口的 ClientNamenodeProtocolTranslatorPB 对象。createNonHAProxy() 方法的代码如下:

```

public static <T> ProxyAndInfo<T> createNonHAProxy(
    Configuration conf, InetSocketAddress nnAddr, Class<T> xface,
    UserGroupInformation ugi, boolean withRetries) throws IOException {
    Text dtService = SecurityUtil.buildTokenService(nnAddr);

    T proxy;
    // 当前接口是 ClientProtocol, 调用对应的方法
    if (xface == ClientProtocol.class) {
        proxy = (T) createNNProxyWithClientProtocol(nnAddr, conf, ugi,
            withRetries);
    } else if (xface == JournalProtocol.class) { // 当前接口是 JournalProtocol
        proxy = (T) createNNProxyWithJournalProtocol(nnAddr, conf, ugi);
    }
    // 其他协议接口 ...
}

```

由于我们一直是以 ClientProtocol 作为例子的, 这里我们就看一下 createNNProxyWithClientProtocol() 方法的实现。createNNProxyWithClientProtocol() 方法首先会设置 RPC.protocolEngine 字段为 ProtobufRpcEngine, RPC.protocolEngine 字段用于指定当前 RPC 调用使用什么序列化方式, 这里配置的 ProtobufRpcEngine 类就定义了当前 RPC 调用是使用 protobuf 作为序列化引擎的。之后 createNNProxyWithClientProtocol() 方法会调用 RPC.getProtocolProxy() 方法获取 ClientNamenodeProtocolPB 协议的代理对象, 然后构造 ClientNamenodeProtocolTranslatorPB 对象并返回。

createNNProxyWithClientProtocol()方法的代码如下：

```
private static ClientProtocol createNNProxyWithClientProtocol(
    InetAddress address, Configuration conf, UserGroupInformation ugi,
    boolean withRetries, AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    // 设置 protocolEngine
    RPC.setProtocolEngine(conf, ClientNamenodeProtocolPB.class, ProtobufRpcEngine.class);
    // ...
    // 构造 ClientNamenodeProtocolPB 代理对象
    ClientNamenodeProtocolPB proxy = RPC.getProtocolProxy(
        ClientNamenodeProtocolPB.class, version, address, ugi, conf,
        NetUtils.getDefaultSocketFactory(conf),
        org.apache.hadoop.ipc.Client.getTimeout(conf), defaultPolicy,
        fallbackToSimpleAuth).getProxy();
    // ...
    // 构造 ClientNamenodeProtocolTranslatorPB 对象并返回
    // 注意 ClientNamenodeProtocolTranslatorPB 会持有一个 ClientNamenodeProtocolPB 对象
    return new ClientNamenodeProtocolTranslatorPB(proxy);
}
```

图 2-10 总结了 createNNProxyWithClientProtocol()调用的全部流程。

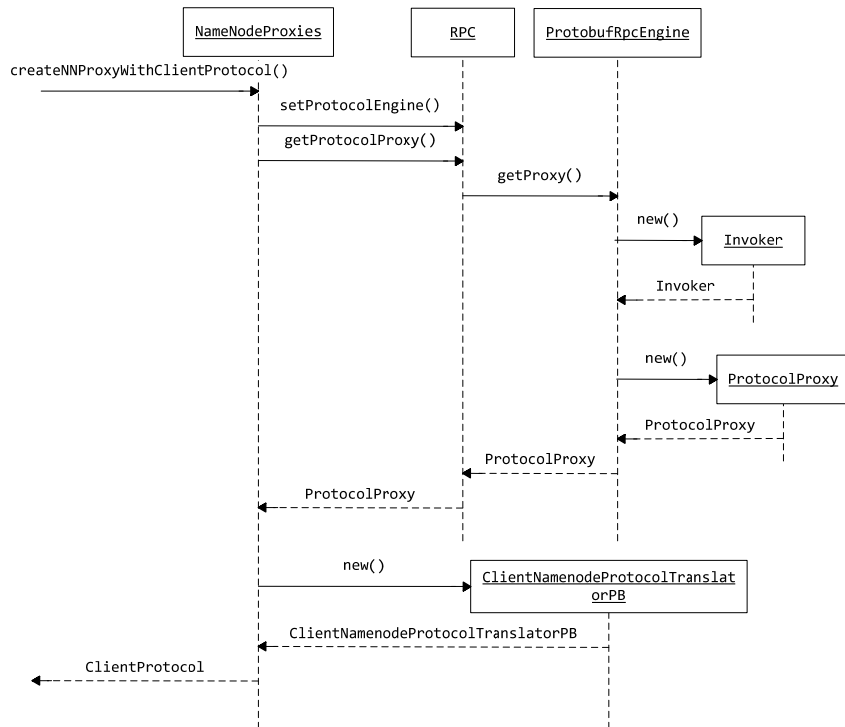


图 2-10 createNNProxyWithClientProtocol()调用流程图

通过调用 `createNNProxyWithClientProtocol()` 方法，我们就成功地获取了一个实现了 `ClientProtocol` 接口的 `ClientNamenodeProtocolTranslatorPB` 对象。下面我们重点看一下获取 `ClientNamenodeProtocolPB` 代理对象的 `RPC.getProtocolProxy()` 方法的实现，这个方法是 `RPC` 类中最重要的一個方法，用于获取一个指定 `RPC` 接口的代理对象。`getProtocolProxy()` 会首先调用 `getProtocolEngine()` 方法获取当前 `RPC` 定义的 `protocolEngine` 对象的实例，然后在这个 `protocolEngine` 对象上调用 `getProxy()` 获取使用特定序列化方式的接口代理对象。`getProtocolProxy()` 方法的代码如下：

```
public static <T> ProtocolProxy<T> getProtocolProxy(Class<T> protocol,
                                                    long clientVersion,
                                                    InetAddress addr,
                                                    UserGroupInformation ticket,
                                                    Configuration conf,
                                                    SocketFactory factory,
                                                    int rpcTimeout,
                                                    RetryPolicy connectionRetryPolicy,
                                                    AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    return getProtocolEngine(protocol, conf).getProxy(protocol, clientVersion,
                                                    addr, ticket, conf, factory, rpcTimeout, connectionRetryPolicy,
                                                    fallbackToSimpleAuth);
}
```

我们看一下 `getProtocolEngine()` 方法的实现。还是以 `ClientProtocol` 为例，由于 `createNNProxyWithClientProtocol()` 的第一条语句就是注册 `ClientProtocol` 的序列化引擎为 `ProtobufRpcEngine`，所以 `ClientProtocol` 使用 `protobuf` 作为序列化工具，那么这里的 `getProtocolEngine()` 方法返回的就是一个 `ProtobufRpcEngine` 的实例。

```
static synchronized RpcEngine getProtocolEngine(Class<?> protocol,
                                                Configuration conf) {
    RpcEngine engine = PROTOCOL_ENGINES.get(protocol);
    // 通过反射创建 rpcEngine 实例
    if (engine == null) {
        Class<?> impl = conf.getClass(ENGINE_PROP+"."+protocol.getName(),
                                    WritableRpcEngine.class);
        engine = (RpcEngine) ReflectionUtils.newInstance(impl, conf);
        PROTOCOL_ENGINES.put(protocol, engine);
    }
    return engine;
}
```

可以看到，真正构造代理对象的方法其实是 `RpcEngine.getProxy()` 方法，这里的 `RpcEngine` 是一个抽象类，它只定义了接口，具体的实现留给子类去做。`RpcEngine` 有两个重要的子类：`WritableRpcEngine` 用于描述使用 `HadoopWritable` 序列化机制的 `RPC` 引擎；而 `ProtobufRpcEngine` 则用于描述使用 `protobuf` 序列化机制的 `RPC` 引擎。之所以这样定义，是为了使 `Hadoop RPC` 可以方便地支持多种 `RPC` 序列化方式，同时在切换 `RPC` 引擎时并不需要更

改源码，只需要通过配置更改 RPC 的 `RpcEngine` 对象即可。我们知道 `ClientProtocol` 默认是使用 `ProtobufRpcEngine` 的，所以我们来看一下 `ProtobufRpcEngine.getProxy()` 方法的实现。如下代码所示，`getProxy()` 方法首先会构造一个实现了 `InvocationHandler` 接口的 `invoker` 对象（动态代理机制中的 `InvocationHandler` 对象会在 `invoke()` 方法中代理所有目标接口上的调用，用户可以在 `invoke()` 方法中添加代理操作），这个 `invoker` 对象是 `ProtoBufRpcEngine.Invoker` 类型的。构造了这个对象之后，`getProxy()` 就可以调用 `Proxy.newProxyInstance()` 方法构造动态代理对象，然后将这个对象封装在 `ProtocolProxy` 对象中并返回。

```
public <T> ProtocolProxy<T> getProxy(Class<T> protocol, long clientVersion,
    InetSocketAddress addr, UserGroupInformation ticket, Configuration conf,
    SocketFactory factory, int rpcTimeout, RetryPolicy connectionRetryPolicy,
    AtomicBoolean fallbackToSimpleAuth) throws IOException {
    // 首先构造 InvocationHandler 对象
    final Invoker invoker = new Invoker(protocol, addr, ticket, conf, factory,
        rpcTimeout, connectionRetryPolicy, fallbackToSimpleAuth);
    // 然后调用 Proxy.newProxyInstance() 获取动态代理对象，并通过 ProtocolProxy 返回
    return new ProtocolProxy<T>(protocol, (T) Proxy.newProxyInstance(
        protocol.getClassLoader(), new Class[]{protocol}, invoker), false);
}
```

由于 Java 动态代理机制决定了在代理对象上的所有调用都会由 `InvocationHandler` 对象的 `invoke()` 方法代理，所以所有在 `ClientNamenodeProtocolPB` 代理对象上的调用都会由这个 `ProtobufRpcEngine.Invoker` 对象的 `invoker()` 方法代理，`ProtobufRpcEngine.Invoker.invoker()` 方法主要做了三件事情：①构造请求头域，使用 `protobuf` 将请求头序列化，这个请求头域记录了当前 RPC 调用是什么接口的什么方法上的调用；②通过 `RPC.Client` 类发送请求头以及序列化好的请求参数。请求参数是在 `ClientNamenodeProtocolPB` 调用时就已经序列化好的，调用 `Client.call()` 方法时，需要将请求头以及请求参数使用一个 `RpcRequestWrapper` 对象封装；③获取响应信息，序列化响应信息并返回。`ProtobufRpcEngine.Invoker` 的 `invoke()` 方法如下：

```
public Object invoke(Object proxy, Method method, Object[] args)
    throws ServiceException {
    // 通过上一节的学习，我们知道 pb 接口的参数只有两个，即 RpcController + Message
    if (args.length != 2) {
        throw new ServiceException("Too many parameters for request. Method: ["
            + method.getName() + "]" + ", Expected: 2, Actual: "
            + args.length);
    }
    if (args[1] == null) {
        throw new ServiceException("null param while calling Method: ["
            + method.getName() + "]");
    }

    // 构造请求头域，标明在什么接口上调用什么方法
    RequestHeaderProto rpcRequestHeader = constructRpcRequestHeader(method);
    // 获取请求调用的参数，例如 RenameRequestProto
    Message theRequest = (Message) args[1];
```



```

final RpcResponseWrapper val;
try {
    // 调用 RPC.Client 发送请求
    val = (RpcResponseWrapper) client.call(RPC.RpcKind.RPC_PROTOCOL_BUFFER,
        new RpcRequestWrapper(rpcRequestHeader, theRequest), remoteId);

} catch (Throwable e) {
    // ...
    throw new ServiceException(e);
}

// ...

Message prototype = null;
try {
    // 获取返回参数类型, RenameResponseProto
    prototype = getReturnProtoType(method);
} catch (Exception e) {
    throw new ServiceException(e);
}
Message returnMessage;
try {
    // 序列化响应信息并返回
    returnMessage = prototype.newBuilderForType()
        .mergeFrom(val.theResponseRead).build();
    // ...
} catch (Throwable e) {
    throw new ServiceException(e);
}
// 返回结果
return returnMessage;
}

```

2. HA 模式

对于 HA 模式, NameNodeProxies 调用 RetryProxy.create()方法构造实现了 RPC 协议的对象, 这里的 RetryProxy 是一个工厂类, 它会构造支持 HA 模式的协议对象, 这个协议对象会首先尝试连接 HDFS 中的 ActiveNamenode, 如果连接失败则会重试, 如果重试达到一定的次数, 则会切换到 HDFS 集群中的 StandbyNamenode。我们直接看一下 RetryProxy 的工厂方法。

```

public static <T> Object create(Class<T> iface,
    FailoverProxyProvider<T> proxyProvider, RetryPolicy retryPolicy) {
    // 直接调用 Java 动态代理构造方法, 返回代理对象
    return Proxy.newProxyInstance(
        proxyProvider.getInterface().getClassLoader(),
        new Class<?>[] { iface },
        new RetryInvocationHandler<T>(proxyProvider, retryPolicy)
    );
}

```

RetryProxy 的工厂方法 RetryProxy()调用了 Java 动态代理方法 Proxy.newProxyInstance(), 那么由上面的代码我们知道, 在协议的代理对象上的调用都会由 RetryInvocationHandler 的 invoke()方法代理(Java 动态代理定义的, 在代理对象上的调用会由 InvocationHandler 的 invoke()方法统一代理)。下面我们就分析一下 RetryInvocationHandler.invoke()方法的实现。invoke()方法的实现主要分为如下几步。

(1) 首先获取 RetryPolicy 对象, RetryPolicy 定义了出现调用错误时的重试逻辑。这里默认的 RetryPolicy 是 failoverOnNetworkException。

(2) 然后 RetryInvocationHandler.invoke()方法通过反射调用 method 对象描述的方法。

```
Object ret = invokeMethod(method, args);
```

invokeMethod()方法其实就是在 currentProxy 对象上调用 method 参数描述的方法——currentProxy 是泛型 T 类型——也就是接口 ClientProtocol 类型。invokeMethod()方法的代码如下:

```
protected Object invokeMethod(Method method, Object[] args) throws Throwable {
    try {
        if (!method.isAccessible()) {
            method.setAccessible(true);
        }
        // 注意, 这里的 currentProxy 会发生切换
        return method.invoke(currentProxy, args);
    } catch (InvocationTargetException e) {
        throw e.getCause();
    }
}
```

currentProxy 是实现了 ClientProtocol 协议的对象, 它是通过调用 FailoverProxyProvider.getProxy()获得的。当 Namenode 发生主从切换后, currentProxy 字段会被赋值为新的 Active Namenode 对应的 ClientProtocol 的引用, 之后在 ClientProtocol 上的调用也就会发送到新的 Active Namenode。

```
this.currentProxy = proxyProvider.getProxy();
```

(3) 如果完成上述操作后没有抛出异常, 也就是说, 客户端成功地将请求发送到了 Active Namenode 服务器。如果抛出异常, 则说明远程调用出现了错误, 这部分代码在 catch 段处理。

■ 通过 annotation 判断这个操作是否是 idempotent (幂等的, 也就是执行多次是没有问题的, 例如 ClientProtocol 中的 setReplication()), 对于幂等的操作可以再次调用。之后调用 RetryPolicy.ShouldRetry()方法分析如何处理这个错误。

```
// 判断方法是否是幂等的
```

```
boolean isIdempotentOrAtMostOnce = proxyProvider.getInterface()
    .getMethod(method.getName(), method.getParameterTypes())
    .isAnnotationPresent(Idempotent.class);
if (!isIdempotentOrAtMostOnce) {
```

```

isIdempotentOrAtMostOnce = proxyProvider.getInterface()
    .getMethod(method.getName(), method.getParameterTypes())
    .isAnnotationPresent(AtMostOnce.class);
}
// 在 RetryPolicy 上分析需要进行的 retry 动作
RetryAction action = policy.shouldRetry(e, retries++,
    invocationFailoverCount, isIdempotentOrAtMostOnce);

```

- 之后调用 `policy.shouldRetry()` 判断是否需要执行重试操作（这里的 `RetryPolicy` 是默认值 `FailoverOnNetworkExceptionRetry`），这里我们分析 `FailoverOnNetworkExceptionRetry.shouldRetry()` 方法的逻辑。
 - 如果失败的次数已经超过最大的次数，就返回一个 `RetryAction.RetryDecision.FAIL` 的 `RetryAction` 表明调用失败。
 - 如果抛出的异常是 `ConnectionException`、`NoRouteToHostException`、`UnknownHostException`、`StandbyException`、`RemoteException` 中的一个，则说明底层的协议代理对象无法连接到 `ActiveNamenode`，或者 `ActiveNamenode` 宕机，或者 HDFS 集群已经发生主从切换了。在这些情况下，就需要返回一个 `RetryAction.RetryDecision.FAILOVER_AND_RETRY` 的 `RetryAction`，表明需要执行 `performFailover()` 操作更新 `Active Namenode` 的引用。
 - 如果抛出的异常是 `SocketException`、`IOException` 或者其他非 `RemoteException` 的异常，则无法判断这个 RPC 命令到底是不是执行成功了。可能是本地的 `Socket` 或者 `IO` 问题，也可能是 `Namenode` 端的 `Socket` 或者 `IO` 问题。这时就进行进一步的判断：如果被调用的方法是 `idempotent`，也就是多次执行是没有副作用的，那么就连接另外一个底层代理重试；否则直接返回 `RetryAction.RetryDecision.FAIL` 表明调用失败。

```

public RetryAction shouldRetry(Exception e, int retries,
    int failovers, boolean isIdempotentOrAtMostOnce) throws Exception {

    // 已经超出了最大重试次数
    if (failovers >= maxFailovers) {
        return new RetryAction(RetryAction.RetryDecision.FAIL, 0,
            "failovers (" + failovers + ") exceeded maximum allowed (" +
                maxFailovers + ")");
    }

    // 判断异常类型
    if (e instanceof ConnectException ||
        e instanceof NoRouteToHostException ||
        e instanceof UnknownHostException ||
        e instanceof StandbyException ||
        e instanceof ConnectTimeoutException ||
        isWrappedStandbyException(e)) {
        return new RetryAction(
            RetryAction.RetryDecision.FAILOVER_AND_RETRY,

```

```
// retry immediately if this is our first failover, sleep otherwise
failovers == 0 ? 0 :
    calculateExponentialTime(delayMillis, failovers, maxDelayBase));
} else if (e instanceof SocketException ||
    (e instanceof IOException && !(e instanceof RemoteException))) {
    if (isIdempotentOrAtMostOnce) {
        return RetryAction.FAILOVER_AND_RETRY;
    } else {
        return new RetryAction(RetryAction.RetryDecision.FAIL, 0,
            "the invoked method is not idempotent, and unable to determine " +
            "whether it was invoked");
    }
} else {
    return fallbackPolicy.shouldRetry(e, retries, failovers,
        isIdempotentOrAtMostOnce);
}
}
```

- 通过异常情况，获得了对应的 `RetryAction` 之后，就会在 `proxyProvider` 上调用 `performFailover()` 方法更新 `currentProxy`。如果是 HA 配置，那么在 Namenode 主从切换后，`performFailover()` 会更新 `currentProxy` 到新的 Active Namenode，然后继续循环，这样在 `currentProxy` 上的调用就可以发送到新的 Active Namenode 了。

```
if (action.action == RetryAction.RetryDecision.FAILOVER_AND_RETRY) {
    synchronized (proxyProvider) {
        if (invocationAttemptFailoverCount == proxyProviderFailoverCount) {
            proxyProvider.performFailover(currentProxy);
            proxyProviderFailoverCount++;
            currentProxy = proxyProvider.getProxy();
        } else {
            LOG.warn()
        }
    }
}
```

2.2.4 服务器获取 Server 对象

在 Hadoop RPC 使用概述小节中我们介绍了 RPC 服务器需要构造一个 `Server` 对象，这个 `Server` 对象用于监听并响应来自 RPC 客户端的请求。例如对于 Namenode，它会构造两个 `Server` 对象分别响应来自 HDFS 客户端和 Datanode 的 RPC 请求。本节就以 Namenode 构造 `Server` 的流程为例，介绍 RPC 服务器端获取 `Server` 对象的代码实现。

1. 构造 NameNodeRpcServer

Namenode 定义了 `NameNodeRpcServer` 类响应来自 HDFS 集群中其他节点的 RPC 请求，如图 2-11 所示，`NameNodeRpcServer` 实现了包括 `ClientProtocol`、`NamenodeProtocol`、`DatanodeProtocol` 以及 `HAServiceProtocol` 在内的所有需要与 Namenode 交互的 RPC 协议接口。

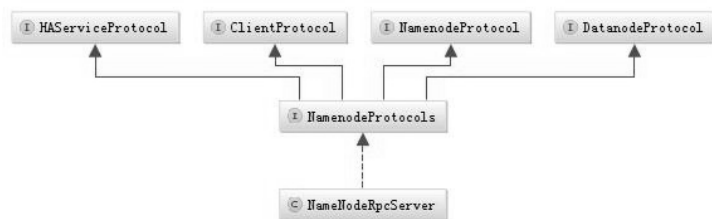


图 2-11 NameNodeRpcServer 实现的接口

NameNode 会在它的初始化方法 `initialize()` 中调用 `createRpcServer()` 创建 `NameNodeRpcServer` 对象的实例，`createRpcServer()` 方法会直接调用 `NameNodeRpcServer` 的构造方法。我们首先看一下这个构造方法。由于 `NameNodeRpcServer` 实现了多个接口，造成它的构造方法非常长，这里只列出了与 `ClientProtocol` 相关的代码，以简化篇幅。

`NameNodeRpcServer` 的构造方法首先设置了 RPC 类的序列化引擎为 `protobuf`，然后构造了两个 `RPC.Server` 对象：`clientRpcServer` 用于响应来自 HDFS 客户端的 RPC 请求；`serviceRpcServer` 则用于响应来自 Datanode 的 RPC 请求。`clientRpcServer` 和 `serviceRpcServer` 的构造方法很类似，都是先调用 `RPC.build()` 方法获取一个 `RPC.Server` 对象，然后调用 `DFSUtil.addPBProtocol()` 方法在新获取的 `RPC.Server` 对象上添加 `*ProtocolPB` 协议与 `BlockingService` 对象之间的映射关系，例如 `ClientNamenodeProtocolPB` 协议会由 `BlockingService` 对象 `clientNNPbService` 处理。之所以配置这个映射关系，是因为当 `RPC.Server` 监听到网络上的 RPC 请求后，它会首先提取出 RPC 请求的请求头域，解析出这次 RPC 请求是在什么接口（接口信息）的什么方法（方法信息）上调用的，然后根据接口信息以及上述配置的映射关系提取出执行响应操作的 `BlockingService` 对象，最后根据方法信息调用 `BlockingService.callBlockingMethod()` 方法响应这个 RPC 调用。所以 `*ProtocolPB` 协议接口与 `BlockingService` 对象之间的映射关系保证了 RPC 请求到达 Server 后，Server 可以找到正确的响应类来执行相应操作。`NameNodeRpcServer` 的构造方法代码如下：

```

public NameNodeRpcServer(Configuration conf, NameNode nn)
    throws IOException {
    this.nn = nn;
    this.namesystem = nn.getNameSystem();
    this.metrics = NameNode.getNameNodeMetrics();

    int handlerCount =
        conf.getInt(DFS_NAMENODE_HANDLER_COUNT_KEY,
            DFS_NAMENODE_HANDLER_COUNT_DEFAULT);

    // 设置 RPC 引擎为 protobuf
    RPC.setProtocolEngine(conf, ClientNamenodeProtocolPB.class,
        ProtobufRpcEngine.class);

    // 构造 ClientNamenodeProtocolServerSideTranslatorPB 对象
    // 这个对象用于适配 ClientProtocolPB 到 ClientProtocol 接口的转换
  
```

Hadoop 2.X HDFS 源码剖析

```
ClientNamenodeProtocolServerSideTranslatorPB
    clientProtocolServerTranslator =
        new ClientNamenodeProtocolServerSideTranslatorPB(this);
// 构造 BlockingService 对象
// 这个对象用于将 Server 提取出的请求前转到 clientProtocolServerTranslator 对象
BlockingService clientNNPbService = ClientNamenodeProtocol.
    newReflectiveBlockingService(clientProtocolServerTranslator);
// ...其他接口同 ClientProtocol 类似
WritableRpcEngine.ensureInitialized();

// 初始化 serviceRpcServer 对象, 这个对象用于响应来自 Datanode 的请求
InetSocketAddress serviceRpcAddr = nn.getServiceRpcServerAddress(conf);
if (serviceRpcAddr != null) {
    String bindHost = nn.getServiceRpcServerBindHost(conf);
    if (bindHost == null) {
        bindHost = serviceRpcAddr.getHostName();
    }
    int serviceHandlerCount =
        conf.getInt(DFS_NAMENODE_SERVICE_HANDLER_COUNT_KEY,
            DFS_NAMENODE_SERVICE_HANDLER_COUNT_DEFAULT);

    // 构造 serviceRpcServer 对象, 并配置 ClientProtocolPB 的响应类为 clientNNPbService
    this.serviceRpcServer = new RPC.Builder(conf)
        .setProtocol(
            org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)
        .setInstance(clientNNPbService)
        .setBindAddress(bindHost)
        .setPort(serviceRpcAddr.getPort()).setNumHandlers(serviceHandlerCount)
        .setVerbose(false)
        .setSecretManager(namesystem.getDelegationTokenSecretManager())
        .build();

    // 注册 NamenodeRPCServer 实现的所有接口
    DFSUtil.addPBProtocol(conf, HAServiceProtocolPB.class, haPbService,
        serviceRpcServer);
    DFSUtil.addPBProtocol(conf, NamenodeProtocolPB.class, NNPbService,
        serviceRpcServer);
    // ...添加其他协议

    // ...更新端口号
} else {
    serviceRpcServer = null;
    serviceRPCAddress = null;
}
InetSocketAddress rpcAddr = nn.getRpcServerAddress(conf);
String bindHost = nn.getRpcServerBindHost(conf);
```

```

if (bindHost == null) {
    bindHost = rpcAddr.getHostName();
}

// 构造 clientRpcServer, 用于响应来自 HDFS 客户端的 RPC 请求
this.clientRpcServer = new RPC.Builder(conf)
    .setProtocol(
        org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)
    .setInstance(clientNNPbService).setBindAddress(bindHost)
    .setPort(rpcAddr.getPort()).setNumHandlers(handlerCount)
    .setVerbose(false)
    .setSecretManager(namesystem.getDelegationTokenSecretManager()).build();

// 注册 NamenodeRpcServer 实现的所有接口
DFSUtil.addPBProtocol(conf, HATServiceProtocolPB.class, haPbService,
    clientRpcServer);
DFSUtil.addPBProtocol(conf, NamenodeProtocolPB.class, NNPbService,
    clientRpcServer);
// ...添加其他协议

// ...安全相关、更新端口号、配置异常处理机制
}

```

图 2-12 总结了 NameNodeRpcServer 构造方法中 serviceRpcServer 的简化版构造流程，整个流程可以分为两个部分：①获取响应 ClientNamenodeProtocolPB 请求的 BlockingService 对象（其他协议的处理流程与 ClientNamenodeProtocolPB 类似）；②构造 RPC.Server。这两部分内容我们会在后面两个小节中分别进行介绍。

2. 获取 BlockingService 对象

在 NameNodeRpcServer 的构造方法中，ClientNamenodeProtocolPB 协议对应的 BlockingService 实现类是通过调用 ClientNamenodeProtocol.newReflectiveBlockingService(BlockingInterfaceImpl)方法获取的，而 newReflectiveBlockingService()方法的 impl 参数则是实现了 ClientNamenodeProtocolPB 接口的 ClientNamenodeProtocolServerSideTranslatorPB 对象，这个对象是直接通过它的构造方法构造的。在 ClientNamenodeProtocolServerSideTranslatorPB 类小节中我们介绍了 ClientNamenodeProtocolServerSideTranslatorPB 对象是将 ClientNamenodeProtocolPB 接口调用适配成 ClientProtocol 调用的适配器类，它内部会持有一个实现了 ClientProtocol 接口的对象，将 ClientNamenodeProtocolPB 调用的参数反序列化之后，它会调用 ClientProtocol 对象的对应方法执行 RPC 操作。通过 NameNodeRpcServer 构造方法的代码我们知道，ClientNamenodeProtocolServerSideTranslatorPB 中持有的 ClientProtocol 接口对象其实就是 NameNodeRpcServer，NameNodeRpcServer 实现了 ClientProtocol 接口，是 Namenode RPC 服务真正的实现类。

Hadoop 2.X HDFS 源码剖析

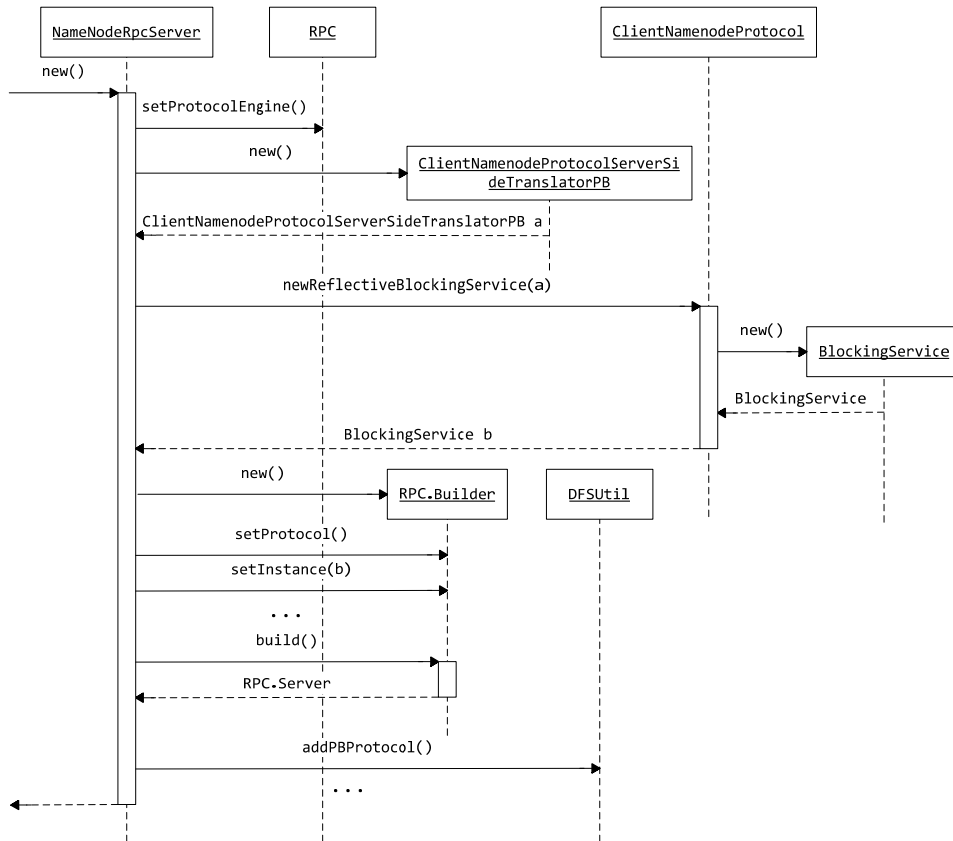


图 2-12 NameNodeRpcServer 构造方法调用流程图

```
public NameNodeRpcServer(Configuration conf, NameNode nn)
    throws IOException {
    // ...

    // 构造 ClientNamenodeProtocolServerSideTranslatorPB 对象
    // 这个对象用于适配 ClientProtocolPB 到 ClientProtocol 接口的转换
    ClientNamenodeProtocolServerSideTranslatorPB
    clientProtocolServerTranslator =
        new ClientNamenodeProtocolServerSideTranslatorPB(this);
    // 构造 BlockingService 对象
    // 这个对象用于将 Server 提取出的请求前转到 clientProtocolServerTranslator 对象
    BlockingService clientNNPbService = ClientNamenodeProtocol.
        newReflectiveBlockingService(clientProtocolServerTranslator);
    // ...
}
```

ClientNamenodeProtocol.newReflectiveBlockingService()方法的定义如下代码所示，它会构

造一个匿名的 `BlockingService` 对象并返回，这个匿名的 `BlockingService` 对象定义了一个 `callBlockingMethod()` 方法。`callBlockingMethod()` 方法接受三个参数：`method` 参数描述了当前 RPC 调用的方法信息；`controller` 参数在这里默认为 `null`，不使用；`request` 参数记录了 RPC 调用的参数信息。`callBlockingMethod()` 方法会根据 `method` 参数记录的调用方法信息，在 `impl` 引用上调用对应的方法。这里的 `impl` 引用是 `ClientNamenodeProtocolServerSideTranslatorPB` 类型的，`ClientNamenodeProtocolServerSideTranslatorPB` 会将 `ClientNamenodeProtocolPB` 调用的参数反序列化，然后前转到 `NamenodeRpcServer` 对象上执行 RPC 操作。这样，`Server` 对象监听到 RPC 请求后，只需通过请求头域中的接口信息获取对应的 `BlockingService` 对象，然后在这个 `BlockingService` 对象上调用 `callBlockingMethod()` 就可以触发 `NameNodeRpcServer` 对象响应这个 RPC 请求了。

```
public static com.google.protobuf.BlockingService
    newReflectiveBlockingService(final BlockingInterface impl) {
return new com.google.protobuf.BlockingService() {
    public final com.google.protobuf.Descriptors.ServiceDescriptor
        getDescriptorForType() {
        return getDescriptor();
    }

    public final com.google.protobuf.Message callBlockingMethod(
        com.google.protobuf.Descriptors.MethodDescriptor method,
        com.google.protobuf.RpcController controller,
        com.google.protobuf.Message request)
        throws com.google.protobuf.ServiceException {
    if (method.getService() != getDescriptor()) {
        throw new java.lang.IllegalArgumentException(
            "Service.callBlockingMethod() given method descriptor for " +
            "wrong service type.");
    }
    switch(method.getIndex()) {
        case 0:
            return impl.getBlockLocations(controller, (GetBlockLocationsRequestProto)
request);
        case 1:
            return impl.getServerDefaults(controller, (GetServerDefaultsRequestProto)
request);
        case 2:
            // ...
        default:
            throw new java.lang.AssertionError("Can't get here.");
    }
    }
    // ...
};
}
```

3. 构造 Server 对象

NameNodeRpcServer 的构造方法调用了 RPC.Server.build()方法构造 Server 对象，build()方法的代码如下所示，它设置了 protocol 参数为 ClientNamenodeProtocolPB.class，同时设置了 impl 参数为 clientNNPbService。这两个参数设置了 ClientNamenodeProtocolPB 协议与它的 BlockingService 响应类的对应关系，也就是说，ClientNamenodeProtocolPB 的调用会由 BlockingService 对象 clientNNPbService 来响应。本节我们就来学习 build()方法的实现，也就是 RPC Server 对象的构造。

```
this.serviceRpcServer = new RPC.Builder(conf)
    .setProtocol(
        org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)
    .setInstance(clientNNPbService)
    .setBindAddress(bindHost)
    .setPort(serviceRpcAddr.getPort()).setNumHandlers(serviceHandlerCount)
    .setVerbose(false)
    .setSecretManager(namesystem.getDelegationTokenSecretManager())
    .build();
```

build()方法首先调用 getProtocolEngine()获取当前 RPC 类配置的 RpcEngine 对象，在 NameNodeRpcServer 的构造方法中已经将当前 RPC 类的 RpcEngine 对象设置为 ProtobufRpcEngine 了。获取了 ProtobufRpcEngine 对象之后，build()方法会在 ProtobufRpcEngine 对象上调用 getServer()方法获取一个 RPC Server 对象的引用。

```
public Server build() throws IOException, HadoopIllegalArgumentException {
    if (this.conf == null) {
        throw new HadoopIllegalArgumentException("conf is not set");
    }
    if (this.protocol == null) {
        throw new HadoopIllegalArgumentException("protocol is not set");
    }
    if (this.instance == null) {
        throw new HadoopIllegalArgumentException("instance is not set");
    }

    return getProtocolEngine(this.protocol, this.conf).getServer(
        this.protocol, this.instance, this.bindAddress, this.port,
        this.numHandlers, this.numReaders, this.queueSizePerHandler,
        this.verbose, this.conf, this.secretManager, this.portRangeConfig);
}
```

在客户端获取 Proxy 对象小节中我们已经介绍了，客户端获取 Proxy 对象是通过调用 RpcEngine.getProxy()方法实现的，对于不同的 RpcEngine 获取的 Proxy 对象是不同的。对于底层来说，就是 RPC 请求所使用的序列化工具是不同的。同理，对于 RPC.Server 来说，不同的 RpcEngine 构造 RPC.Server 对象也是使用不同的反序列化工具，ProtobufRpcEngine.getServer()会返回使用 protobuf 作为反序列化工具的服务器，这个 RPC.Server 对象是 ProtobufRpcEngine 的内部类 Server（RPC.Server 的子类），getServer()方法会构造这个对象并返回。

```

public RPC.Server getServer(Class<?> protocol, Object protocolImpl,
    String bindAddress, int port, int numHandlers, int numReaders,
    int queueSizePerHandler, boolean verbose, Configuration conf,
    SecretManager<? extends TokenIdentifier> secretManager,
    String portRangeConfig)
    throws IOException {
    return new Server(protocol, protocolImpl, conf, bindAddress, port,
        numHandlers, numReaders, queueSizePerHandler, verbose, secretManager,
        portRangeConfig);
}

```

下面我们看一下 `ProtobufRpcEngine.Server` 的构造方法。`ProtobufRpcEngine.Server` 是 `RPC.Server` 的子类，它的构造方法首先调用父类 `RPC.Server` 的构造方法（`RPC.Server` 类的实现我们将在下一节中介绍），之后构造方法会调用 `registerProtocolAndImpl()` 方法注册接口类 `protocolClass` 和实现类 `protocolImpl` 的映射关系。这样，当客户端的 RPC 请求到达时，就可以通过这个映射关系获得具体的实现类了。

```

public Server(Class<?> protocolClass, Object protocolImpl,
    Configuration conf, String bindAddress, int port, int numHandlers,
    int numReaders, int queueSizePerHandler, boolean verbose,
    SecretManager<? extends TokenIdentifier> secretManager,
    String portRangeConfig)
    throws IOException {
    super(bindAddress, port, null, numHandlers,
        numReaders, queueSizePerHandler, conf, classNameBase(protocolImpl
            .getClass().getName()), secretManager, portRangeConfig);
    this.verbose = verbose;
    registerProtocolAndImpl(RPC.RpcKind.RPC_PROTOCOL_BUFFER, protocolClass,
        protocolImpl);
}

```

`ProtobufRpcEngine.Server` 类最重要的部分就是实现了一个 `ProtoBufRpcInvoker` 类，当 `RPC.Server` 类解析来自网络的 RPC 请求后，会调用 `ProtoBufRpcInvoker.call()` 方法响应这个请求。这里我们重点看一下 `call()` 方法的实现。`call()` 方法首先会从请求头中提取出 RPC 调用的接口名和方法名等信息，然后根据调用的接口信息获取对应的 `BlockingService` 对象，再根据调用的方法信息在 `BlockingService` 对象上调用 `callBlockingMethod()` 方法并将调用前转到 `ClientNamenodeProtocolServerSideTranslatorPB` 对象上，最终这个请求会由 `NameNodeRpcServer` 响应。

```

public Writable call(RPC.Server server, String protocol,
    Writable writableRequest, long receiveTime) throws Exception {
    // 获取 rpc 调用头
    RpcRequestWrapper request = (RpcRequestWrapper) writableRequest;
    RequestHeaderProto rpcRequest = request.requestHeader;
    // 获得调用的接口名、方法名、版本号
    String methodName = rpcRequest.getMethodName();
    String protoName = rpcRequest.getDeclaringClassProtocolName();
    long clientVersion = rpcRequest.getClientProtocolVersion();
}

```

Hadoop 2.X HDFS 源码剖析

```
// 获得该接口在 Server 侧对应的实现类
ProtoClassProtoImpl protocolImpl = getProtocolImpl(server, protoName,
    clientVersion);
BlockingService service = (BlockingService) protocolImpl.protocolImpl;
MethodDescriptor methodDescriptor = service.getDescriptorForType()
    .findMethodByName(methodName);
if (methodDescriptor == null) {
    String msg = "Unknown method " + methodName + " called on " + protocol
        + " protocol.";
    LOG.warn(msg);
    throw new RpcNoSuchMethodException(msg);
}
// 获取调用的方法描述符以及调用参数
Message prototype = service.getRequestPrototype(methodDescriptor);
Message param = prototype.newBuilderForType()
    .mergeFrom(request.theRequestRead).build();

Message result;
try {
    // ...
    // 在实现类上调用 callBlockingMethod 方法，级联适配调用到 NameNodeRpcServer
    result = service.callBlockingMethod(methodDescriptor, null, param);
    // ...
} catch (ServiceException e) {
    throw (Exception) e.getCause();
} catch (Exception e) {
    throw e;
}
return new RpcResponseWrapper(result);
}
```

通过对 `call()` 方法的分析我们知道，客户端在什么接口上调用什么方法是在请求头 `requestHeader` 中保存的，`ProtoBufRpcInvoker` 获取了接口信息之后，会调用 `getProtocolImpl()` 方法通过接口名获取对应的实现类。这个映射信息是在 `RPC.Server` 构造时创建的，同时也可以 在 `NameNodeRpcServer` 的构造方法中调用 `DFSUtil.addPBProtocol()` 方法添加。
`getProtocolImpl()` 方法的实现如下：

```
private static ProtoClassProtoImpl getProtocolImpl(RPC.Server server,
    String protoName, long clientVersion) throws RpcServerException {
    ProtoNameVer pv = new ProtoNameVer(protoName, clientVersion);
    // 从 RPC.Server 的 ProtocolImplMap 对象中获取接口信息对应的实现类
    ProtoClassProtoImpl impl =
        server.getProtocolImplMap(RPC.RpcKind.RPC_PROTOCOL_BUFFER).get(pv);
    if (impl == null) { //
        VerProtocolImpl highest =
            server.getHighestSupportedProtocol(RPC.RpcKind.RPC_PROTOCOL_BUFFER,
                protoName);
        // 如果不存在实现类，则抛出异常
    }
}
```

```

    if (highest == null) {
        throw new RpcNoSuchProtocolException(
            "Unknown protocol: " + protoName);
    }
    // 如果 RPC 版本不匹配, 则抛出异常
    throw new RPC.VersionMismatch(protoName, clientVersion,
        highest.version);
}
// 返回实现类
return impl;
}

```

通过接口信息获取对应的 `BlockingService` 实现对象后, `call()` 方法会在这个 `BlockingService` 对象上调用 `BlockingService.callBlockingMethod()` 方法, 这部分内容我们在获取 `BlockingService` 对象小节中已经介绍过了。至此, 一个完整的 `Server` 对象构造以及响应 RPC 请求的流程就介绍完了。

2.3 Hadoop RPC 实现

Hadoop RPC 框架主要由三个类组成: `RPC`、`Client` 和 `Server` 类。`RPC` 类用于对外提供一个使用 Hadoop RPC 框架的接口, `Client` 类用于实现 RPC 客户端功能, `Server` 类则用于实现 RPC 服务器端功能。本节分别介绍这三个类的代码实现。

2.3.1 RPC 类实现

`RPC` 类为使用 Hadoop RPC 框架的代码提供了一个统一的接口, 同时隐藏了底层 RPC 通信的实现细节, 方便了用户的使用。

如图 2-13 所示, 客户端调用程序可以通过调用 `RPC` 类提供的 `waitForProxy()` 和 `getProxy()` 方法获取指定 RPC 协议的代理对象, 之后 `RPC` 客户端就可以调用代理对象的方法发送 RPC 请求到服务器了。`getProxy()` 方法的实现我们在客户端获取 `Proxy` 对象小节中已经介绍过了, 这里不再重复介绍。

而在服务器侧, 服务程序会调用 `RPC` 内部类 `Builder.build()` 方法构造一个 `RPC.Server` 类, 然后调用 `RPC.Server.start()` 方法启动 `Server` 对象监听并响应 RPC 请求。这里的 `RPC.Builder` 内部类是 `RPC` 定义的用来构造 `RPC.Server` 对象的工厂类, 用户可以调用 `Builder.set*()` 方法对 `RPC.Server` 对象进行配置, 之后调用 `build()` 方法构造这个 `RPC.Server` 对象。`RPC.Server` 内部类则是 `Server` 的子类, 它将监听到的 RPC 请求委托给了 `RPC` 内部接口 `RpcInvoker` 的子类处理, 当 `RPC.Server` 监听到一个 RPC 请求后, 它会调用 `RpcInvoker.call()` 方法处理这个请求。这部分内容我们在服务器获取 `Server` 对象小节中也介绍过了, 这里不再重复介绍。

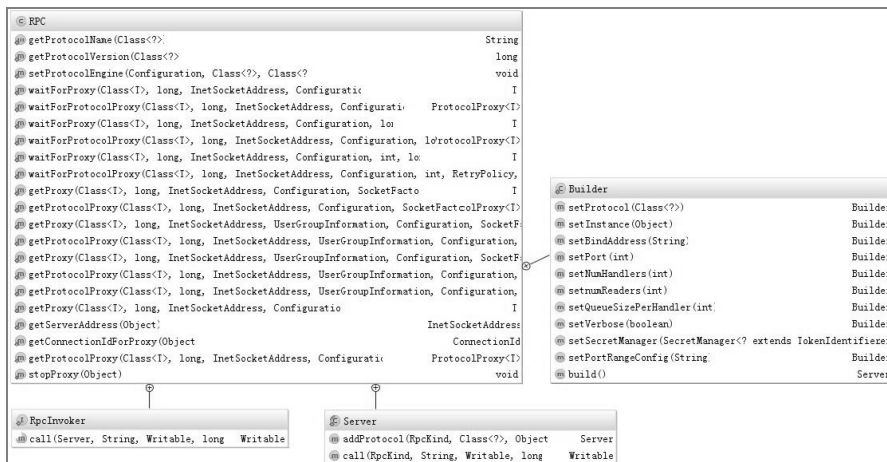


图 2-13 RPC 类结构图

RPC 类还提供了 `setProtocolEngine()` 方法用于配置 RPC 框架当前使用的序列化引擎，以及 `getProtocolEngine()` 方法用于获取序列化引擎对象。在客户端获取 Proxy 对象以及服务器获取 Server 对象小节中我们介绍了 `RPC.getProtocolProxy()` 以及 `RPC.Builder.build()` 方法都会先调用 `getProtocolEngine()` 获取当前 RPC 配置的序列化引擎，然后调用 `RpcEngine.getProxy()` 以及 `RpcEngine.getServer()` 方法执行构造 Proxy 对象和 Server 对象操作。这里以 `getProtocolProxy()` 方法为例，`RPC.Builder.build()` 方法就不再贴出代码了。

```
public static <T> ProtocolProxy<T> getProtocolProxy(Class<T> protocol,
                                                    long clientVersion,
                                                    InetAddress addr,
                                                    UserGroupInformation ticket,
                                                    Configuration conf,
                                                    SocketFactory factory,
                                                    int rpcTimeout,
                                                    RetryPolicy connectionRetryPolicy,
                                                    AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    // 首先调用 getProtocolEngine() 方法获取 RPC 引擎
    return getProtocolEngine(protocol, conf).getProxy(protocol, clientVersion,
                                                    addr, ticket, conf, factory, rpcTimeout, connectionRetryPolicy,
                                                    fallbackToSimpleAuth);
}
```

2.3.2 Client 类实现

在 Hadoop RPC 的使用小节中我们介绍了 `DFSClient` 会获取一个 `ClientProtocolPB` 协议的代理对象，并在这个代理对象上调用 RPC 方法，代理对象会调用 `RPC.Client.call()` 方法将序列化之后的 RPC 请求发送到服务器。本节就介绍 Client 类的实现。

1. Client 发送请求与接收响应流程

Client 类只有一个入口，就是 `call()` 方法。代理类会调用 `Client.call()` 方法将 RPC 请求发送到远程服务器，然后等待远程服务器的响应。如果远程服务器响应请求时出现异常，则在 `call()` 方法中抛出异常。`Client.call()` 方法发送请求与接收响应的流程如图 2-14 所示，分为以下几步。

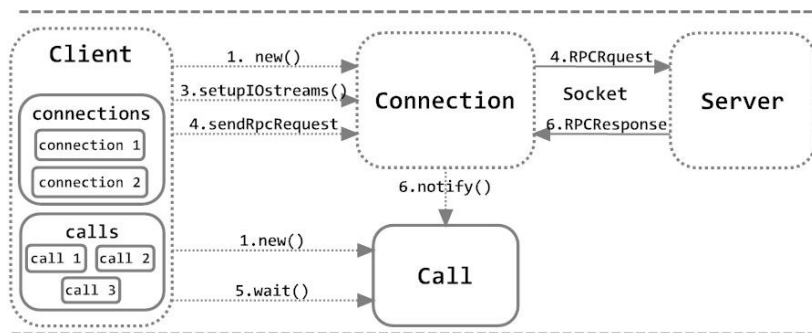


图 2-14 Client.call()方法执行流程图

- Client.call()方法将 RPC 请求封装成一个 Call 对象，Call 对象中保存了 RPC 调用的完成标志、返回值信息以及异常信息；随后，Client.call()方法会创建一个 Connection 对象，Connection 对象用于管理 Client 与 Server 的 Socket 连接。
- 用 ConnectionId 作为 key，将新建的 Connection 对象放入 Client.connections 字段中保存（对于 Connection 对象，由于涉及了与 Server 建立 Socket 连接，会比较耗费资源，所以 Client 类使用一个 HashTable 对象 connections 保存那些没有过期的 Connection，如果可以复用，则复用这些 Connection 对象）；以 callId 作为 key，将构造的 Call 对象放入 Connection.calls 字段中保存。
- Client.call()方法调用 Connection.setupIOstreams()方法建立与 Server 的 Socket 连接。setupIOstreams()方法还会启动 Connection 线程，Connection 线程会监听 Socket 并读取 Server 发回的响应信息。
- Client.call()方法调用 Connection.sendRpcRequest()方法发送 RPC 请求到 Server。
- Client.call()方法调用 Call.wait()在 Call 对象上等待，等待 Server 发回响应信息。
- Connection 线程收到 Server 发回的响应信息，根据响应消息中携带的信息找到对应的 Call 对象，然后设置 Call 对象的返回值字段，并调用 call.notify()唤醒调用 Client.call()方法的线程读取 Call 对象的返回值。

Client.call()方法的代码如下：

```

public Writable call(RPC.RpcKind rpcKind, Writable rpcRequest,
    ConnectionId remoteId, int serviceClass,
    AtomicBoolean fallbackToSimpleAuth) throws IOException {
    // 构造 Call 对象
    final Call call = createCall(rpcKind, rpcRequest);
    // 构造 Connection 对象
    Connection connection = getConnection(remoteId, call, serviceClass,

```

Hadoop 2.X HDFS 源码剖析

```
        fallbackToSimpleAuth);
    try {
        connection.sendRpcRequest(call); // 发送 RPC 请求
    } catch (RejectedExecutionException e) {
        throw new IOException("connection has been closed", e);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new IOException(e);
    }

    boolean interrupted = false;
    synchronized (call) {
        while (!call.done) {
            try {
                call.wait(); // 等待 RPC 响应
            } catch (InterruptedException ie) {
                interrupted = true;
            }
        }

        if (interrupted) {
            Thread.currentThread().interrupt();
        }

        if (call.error != null) { // 发送线程被唤醒，但是服务器处理 RPC 请求时出现异常
            // 从 Call 对象中获取异常，并抛出
            if (call.error instanceof RemoteException) {
                call.error.fillInStackTrace();
                throw call.error;
            } else {
                InetAddress address = connection.getRemoteAddress();
                throw NetUtils.wrapException(address.getHostAddress(),
                    address.getPort(),
                    NetUtils.getHostname(),
                    0,
                    call.error);
            }
        } else {
            // 服务器成功发回响应信息，返回 RPC 响应
            return call.getRpcResponse();
        }
    }
}
```

可以看到 `RPC.Client` 的实现涉及很多内部类，例如 `Call`、`ConnectionId`、`Connection` 等，我们首先介绍一下这些内部类的实现，然后再详细介绍发送请求以及接收响应这两个流程的实现。

2. 内部类——Call

RPC.Client 中发送请求和接收响应是由两个独立的线程进行的，发送请求线程就是调用 Client.call()方法的线程，而接收响应线程则是 call()启动的 Connection 线程。那么这两个线程是如何同步 Server 发回的响应信息的呢？这里就使用了 Call 类。举个例子，线程 1 调用 Client.call()发送 RPC 请求到 Server，然后在这个请求对应的 Call 对象上调用 Call.wait()方法等待 Server 发回响应信息。当线程 2 从 Server 接收了响应信息后，会设置 Call.rpcResponse 字段保存响应信息，然后调用 Call.notify()方法唤醒线程 1。线程 1 被唤醒后，会取出 Call.rpcResponse 字段中记录的 Server 发回的响应信息并返回。Call 对象巧妙地同步了 RPC 请求的发送线程以及 RPC 响应的接收线程，值得我们积累和学习。

Call 对象标识了一个 RPC 请求，它定义了如下字段用于记录一个 RPC 请求的信息。

```
final int id;           // RPC 请求的 id
final int retry;        // 重试次数
final Writable rpcRequest; // 序列化的 RPC 请求
Writable rpcResponse;   // 序列化的 RPC 响应，如果调用发生错误，则这个字段为空
IOException error;       // 如果发生错误，则保存远程的异常
final RPC.RpcKind rpcKind; // Rpc 引擎类型
boolean done;           // 调用操作是否完成
```

下面我们看一下 Call 的方法实现。当 Server 成功地执行了 RPC 调用，并发回响应到接收线程后，接收线程会调用 Call.setRpcResponse()方法保存 Server 发回的响应信息。如果 Server 在执行 RPC 调用时出现异常，则接收线程会调用 Call.setException()方法保存异常信息。要特别注意的是，setRpcResponse()以及 setException()都会唤醒在 Call 对象上等待的请求发送线程。

```
public synchronized void setException(IOException error) {
    this.error = error; // 保存异常信息
    callComplete();    // 调用 callComplete()方法唤醒在 Call 对象上等待的线程
}

public synchronized void setRpcResponse(Writable rpcResponse) {
    this.rpcResponse = rpcResponse; // 保存响应信息
    callComplete();    // 调用 callComplete()方法唤醒在 Call 对象上等待的线程
}

protected synchronized void callComplete() {
    this.done = true; // 设置 done 字段为 true，表明当前请求
    notify();        // 唤醒在 Call 对象上等待的线程
}
```

3. 内部类——Connection

内部类 Connection 是一个线程类，它提供了建立 Client 到 Server 的 Socket 连接、发送 RPC 请求以及读取 RPC 响应信息等功能。Connection 的字段多是与网络连接相关的，如 Socket 输入输出流、超时时间、重发次数等。Connection 类定义的字段如下：

```
private InetSocketAddress server; // Server IP 端口
private final ConnectionId remoteId; // connectionId 唯一标识一个 Connection
// ...
private Socket socket = null; // 到 Server 的 Socket 连接
private DataInputStream in;
private DataOutputStream out;
private int rpcTimeout;
private int maxIdleTime; // 最长空闲时间

private Hashtable<Integer, Call> calls = new Hashtable<Integer, Call>(); // 使用这个
Connection 对象发送的请求
private AtomicBoolean shouldCloseConnection = new AtomicBoolean(); // 是否关闭这个连接
private IOException closeException; // 导致连接关闭的异常
```

- 网络相关字段：与 Server 进行网络通信的所有字段，例如 socket、inputStream、outputStream、maxIdleTime、tcpNoDelay 等。
- calls 字段：一个 Connection 对象往往对应于到一个 Server 的 Socket 连接，在这个 Server 上的所有 RPC 调用是可以共用这个 Connection 对象的，所以 Connection 定义了 calls 字段保存这些调用对应的 Call 对象。
- shouldCloseConnection 字段：是否关闭当前 Connection 对象的标识符。

对于 Connection 类的分析我们还是从入口方法切入，RPC.Client.call()方法会首先调用 Connection.getConnection()方法获取一个 Connection 对象。getConnection()方法首先尝试从 RPC.Client.connections 字段中提取缓存的 Connection 对象。RPC.Client.connections 字段是 Hashtable(Connectionid -> Connection)类型的，用于将已经成功建立 Socket 连接的 Connection 对象缓存起来，这是因为 Connection 类中与 Server 建立 Socket 连接的过程非常耗费资源，所以 Client 类使用 connections 字段保存那些没有过期的 Connection 对象，如果可以复用，则复用这些 Connection 对象。如果 RPC.Client.connections 字段没有缓存 Connection 对象，则 getConnection()方法会直接调用 Connection 的构造方法创建新的 Connection 对象，并将新构造的 Connection 对象放入 RPC.Client.connections 字段中保存。成功地获取了 Connection 对象后，getConnection()方法会调用 addCall()方法将待发送的 RPC 请求对象 Call 添加到这个 Connection 的请求队列 calls 当中，然后调用 setupIOstreams()方法初始化到 Server 的 Socket 连接并获取 IO 流。getConnection()方法的代码如下：

```
private Connection getConnection(ConnectionId remoteId,
    Call call, int serviceClass, AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    if (!running.get()) {
        throw new IOException("The client is stopped");
    }
    Connection connection;
    do {
        synchronized (connections) {
            // 首先尝试从 Client.connections 队列中获取 Connection 对象
            connection = connections.get(remoteId);
            if (connection == null) {
```

```

        // 如果 connections 队列中没有保存, 则构造新的对象
        connection = new Connection(remoteId, serviceClass);
        connections.put(remoteId, connection);
    }
}
// 将待发送请求对应的 Call 对象放入 Connection.calls 队列
} while (!connection.addCall(call));

// 调用 setupIOStreams() 方法, 初始化 Connection 对象并获取 IO 流
connection.setupIOStreams(fallbackToSimpleAuth);
return connection;
}

```

Connection 的构造方法以及 setupIOStreams() 方法的实现我们在下面两个小节中分析。

(1) Connection 构造方法

Connection 类的构造方法比较简单, 它会根据传入的 ConnectionId 对 Connection 对象中的字段赋值, 同时由于 Connection 还是一个线程类, 构造方法还会将当前线程设置为精灵线程。

(2) setupIOStreams() 方法

可以看到 Connection 类的构造方法并没有建立到远程服务器的 Socket 连接, 这个部分是由 setupIOStreams () 方法执行的。setupIOStreams () 方法除了连接到远程服务器并建立 IO 流外, 还会向服务器发送一个连接头, 然后启动 Connection 线程监听 Socket 输入流并等待服务器返回 RPC 响应。由于 setupIOStreams () 方法很长, 我们将其拆分成几个部分进行介绍。

- 首先调用 setupConnection() 建立到 Server 的 Socket 连接, 并且在这个 Socket 连接上获得 InputStream 和 OutputStream 对象。
- 调用 writeConnectionHeader() 方法在连接建立时发送连接头域。
- 将 inputStream 和 outputStream 装饰成 DataInputStream 和 DataOutputStream 类型, 方便以后读写。
- 调用 writeConnectionContext() 方法写入连接上下文。
- 调用 touch() 方法更新上次活跃时间。
- 调用 start() 方法启动 Connection 线程监听并接收 Server 发回的响应信息。

```

private synchronized void setupIOStreams(
    AtomicBoolean fallbackToSimpleAuth) {
    try {
        short numRetries = 0;
        Random rand = null;
        while (true) {
            // 1. 调用 setupConnection() 方法建立到远程服务器的连接
            setupConnection();
            InputStream inStream = NetUtils.getInputStream(socket);
            OutputStream outStream = NetUtils.getOutputStream(socket);

```

Hadoop 2.X HDFS 源码剖析

```
// 2. 发送连接头域
writeConnectionHeader(outStream);

// 3. 包装输入输出流
if (doPing) {
    inStream = new PingInputStream(inStream);
}
this.in = new DataInputStream(new BufferedInputStream(inStream));
if (!(outStream instanceof BufferedOutputStream)) {
    outStream = new BufferedOutputStream(outStream);
}
this.out = new DataOutputStream(outStream);

// 4. 写入连接上下文头域
writeConnectionContext(remoteId, authMethod);

// 5. 更新上次活跃时间
touch();

// 6. 启动 Connection 线程
start();
return;
}
} catch (Throwable t) {
    if (t instanceof IOException) {
        markClosed((IOException)t);
    } else {
        markClosed(new IOException("Couldn't set up IO streams", t));
    }
    close();
}
}
```

(3) 发送请求——Connection.sendRpcRequest()

RPC 发送请求线程会调用 Connection.sendRpcRequest()方法发送 RPC 请求到 Server，这里要特别注意，这个方法不是由 Connection 线程调用的，而是由发起 RPC 请求的线程调用的。我们知道客户端获取的协议代理类会将请求元数据（保存调用的接口信息和方法信息）以及请求参数封装到一个 Writable 对象中，然后调用 Client.call()方法将这个 Writable 类型的请求发送到 Server 端。sendRpcRequest()方法还会发送一个 RPC 请求头给 Server，这个 RPC 请求头保存了本次 RPC 请求的序列化类型、请求 id、请求重试次数等信息。Server 在处理这次 RPC 请求时会根据 RPC 请求头域中的信息配置请求处理的流程，同时在发回 RPC 响应时携带请求 id 等信息。Connection 对象收到响应信息后，会提取出请求 id，然后根据请求 id 查找本次响应对应的 Call 对象，最后执行保存响应信息的操作。sendRpcRequest()方法的执行流程如下：

```
public void sendRpcRequest(final Call call)
    throws InterruptedException, IOException {
```

```

if (shouldCloseConnection.get()) {
    return;
}

// 先构造 RPC 请求头
final DataOutputBuffer d = new DataOutputBuffer();
RpcRequestHeaderProto header = ProtoUtil.makeRpcRequestHeader(
    call.rpcKind, OperationProto.RPC_FINAL_PACKET, call.id, call.retry,
    clientId);
// 将 RPC 请求头写入输出流
header.writeDelimitedTo(d);
// 将 RPC 请求（包括请求元数据和请求参数）写入输出流
call.rpcRequest.write(d);

// 这里使用线程池将请求发送出去，请求包括三个部分：① 长度；② RPC 请求头；③ RPC 请求（包括请求元
// 数据以及请求参数）
synchronized (sendRpcRequestLock) {
    Future<?> senderFuture = SEND_PARAMS_EXECUTOR.submit(new Runnable() {
        @Override
        public void run() {
            try {
                synchronized (Connection.this.out) {
                    if (shouldCloseConnection.get()) {
                        return;
                    }

                    byte[] data = d.getData();
                    int totalLength = d.getLength();
                    out.writeInt(totalLength); // 总长度
                    out.write(data, 0, totalLength); // RPC 请求头 + RPC 请求（请求元数据+参数）
                    out.flush();
                }
            } catch (IOException e) {
                // 如果发生发送异常，则直接关闭连接
                markClosed(e);
            } finally {
                // 之前申请的 buffer 给关闭了，比较优雅
                IOUtils.closeStream(d);
            }
        }
    });

    // 获取执行结果
    try {
        senderFuture.get();
    } catch (ExecutionException e) {
        Throwable cause = e.getCause();
        // 如果有异常则直接抛出
    }
}

```

```

    if (cause instanceof RuntimeException) {
        throw (RuntimeException) cause;
    } else {
        throw new RuntimeException("unexpected checked exception", cause);
    }
}
}
}

```

Client 向 Server 发送的一个完整的 RPC 请求格式如图 2-15 所示。

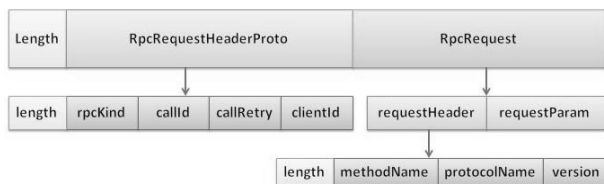


图 2-15 RPC 请求格式

- **length**: 每个 protobuf 类型的数据都包含一个 **length** 字段，这是因为，在 HDFS 写入操作时，使用了 `writeDelimitedTo()` 方法。这个方法会先写入数据的 **length**，然后再写入数据。
- **RpcRequestHeaderProto**: RPC 调用头域，保存了 **callId**、**clientId**、**rpcKind** 等重要信息。服务器发回的响应消息中会带回 **clientId**、**callId** 等信息，用于提取 **call**、鉴权等。
- **RpcRequest**: 要特别注意，这里的 **RpcRequest** 是在 `ProtobufRpcEngine.Invoker.invoke()` 方法中构造的 `RpcRequestWrapper` 类。其中包括两个部分。
 - **requestHeader**: 请求元信息，在什么接口上调用什么方法。例如在 `ClientProtocol` 接口上调用了 `rename()` 方法。
 - **requestParam**: 请求参数，使用 `protobuf` 包装的，例如 `rename()` 请求的 `RenameRequestProto` 参数。

(4) 接收响应——`Connection.run()`

`Connection` 线程负责监听并接收从 Server 发回的 RPC 响应。我们看一下 `Connection.run()` 方法的实现。`Connection.run()` 方法会调用 `waitForWork()` 等待执行读取操作，等待结束后调用 `receiveRpcResponse()` 方法接收 RPC 响应。

```

public void run() {
    try {
        // 调用 waitForWork() 等待
        while (waitForWork()) {
            // 接收响应
            receiveRpcResponse();
        }
    } catch (Throwable t) {
        // 异常都在 receiveRpcResponse 中捕获，这里捕获未知的异常
    }
}

```

```

        markClosed(new IOException("Error reading responses", t));
    }
    // 关闭连接
    close();
}

```

run()方法调用了 waitForWork()方法等待读取操作，waitForWork()方法会判断当前 Connection 的 calls 队列当中是否有 Call 对象，如果没有则等待 maxTimeOut 时长（calls 队列没有对象则表明 Connection 空闲，没有待发送请求）。等待之后如果还没有数据，则返回 false，并且关闭 Connection 对象。

```

private synchronized boolean waitForWork() {
    if (calls.isEmpty() && !shouldCloseConnection.get() && running.get()) {
        long timeout = maxIdleTime-
            (Time.now()-lastActivity.get());
        if (timeout>0) {
            try {
                wait(timeout);
            } catch (InterruptedException e) {}
        }
    }

    if (!calls.isEmpty() && !shouldCloseConnection.get() && running.get()) {
        return true;
    } else if (shouldCloseConnection.get()) {
        return false;
    } else if (calls.isEmpty()) { // 空闲的连接，则关闭
        markClosed(null);
        return false;
    } else { // 仍有等待处理的请求，但是连接被关闭了
        markClosed((IOException)new IOException().initCause(
            new InterruptedException()));
        return false;
    }
}

```

等待结束后，调用 receiveRpcResponse()方法接收 RPC 响应。receiveRpcResponse()方法会从输入流中读取序列化对象 RpcResponseHeaderProto，然后根据 RpcResponseHeaderProto 中记录的 callid 字段获取对应的 Call 的对象。接下来 receiveRpcResponse()方法会从输入流中读取响应消息，然后调用 Call.setRpcResponse()将响应消息保存在 Call 对象中。如果服务器在处理 RPC 请求时抛出异常，则 receiveRpcResponse()会从输入流中读取异常信息，并构造异常对象，然后调用 Call.setException()将异常保存在 Call 对象中。

```

private void receiveRpcResponse() {
    // ...

    try {
        int totalLen = in.readInt();
    }
}

```

Hadoop 2.X HDFS 源码剖析

```
RpcResponseHeaderProto header =
    RpcResponseHeaderProto.parseDelimitedFrom(in);
checkResponse(header);

int headerLen = header.getSerializedSize();
headerLen += CodedOutputStream.computeRawVarint32Size(headerLen);

int callId = header.getCallId();

Call call = calls.get(callId);
RpcStatusProto status = header.getStatus();

// 如果调用成功, 则读取响应消息, 在 call 实例中设置
if (status == RpcStatusProto.SUCCESS) {
    Writable value = ReflectionUtils.newInstance(valueClass, conf);
    value.readFields(in);           // 读取响应消息
    calls.remove(callId);
    call.setRpcResponse(value);

    // ....
}
} else { // RPC 调用失败
    if (totalLen != headerLen) {
        throw new RpcClientException(
            "RPC response length mismatch on rpc error");
    }

    // 取出响应中的异常消息
    final String exceptionClassName = header.hasExceptionClassName() ?
        header.getExceptionClassName() :
            "ServerDidNotSetExceptionClassName";
    final String errorMsg = header.hasErrorMsg() ?
        header.getErrorMsg() : "ServerDidNotSetErrorMsg" ;
    final RpcErrorCodeProto erCode =
        (header.hasErrorDetail() ? header.getErrorDetail() : null);
    // 构造异常
    RemoteException re =
        ( (erCode == null) ?
            new RemoteException(exceptionClassName, errorMsg) :
            new RemoteException(exceptionClassName, errorMsg, erCode));
    // 在 Call 对象中设置异常
    if (status == RpcStatusProto.ERROR) {
        calls.remove(callId);
        call.setException(re);
    } else if (status == RpcStatusProto.FATAL) {
        // Close the connection
        markClosed(re);
    }
}
```



```

    }
} catch (IOException e) {
    markClosed(e);
}
}
}

```

保存好响应消息或者异常之后，在 `Call` 对象上等待的发送请求线程会被唤醒，`Client.call()` 方法会从 `Call` 对象中取出保存的响应消息并返回，如果 `Call` 对象中保存的是异常，则直接抛出异常。`Client.call()` 方法中与响应消息处理相关的代码如下：

```

public Writable call(RPC.RpcKind rpcKind, Writable rpcRequest,
    ConnectionId remoteId, int serviceClass,
    AtomicBoolean fallbackToSimpleAuth) throws IOException {

    // 发送请求

    //
    boolean interrupted = false;
    synchronized (call) {
        while (!call.done) {
            try {
                call.wait(); // 等待结果
            } catch (InterruptedException ie) {
                interrupted = true;
            }
        }

        // 提取出 Call 对象中保存的异常，直接抛出
        if (call.error != null) {
            if (call.error instanceof RemoteException) {
                call.error.fillInStackTrace();
                throw call.error;
            } else {
                InetAddress address = connection.getRemoteAddress();
                throw NetUtils.wrapException(address.getHostName(),
                    address.getPort(),
                    NetUtils.getHostname(),
                    0,
                    call.error);
            }
        } else { // 如果成功执行，则返回 Call 对象中保存的 RPC 响应消息
            return call.getRpcResponse();
        }
    }
}

```

至此，一个 `Client` 发送 RPC 请求以及接收 RPC 响应的流程我们就介绍完了。

2.3.3 Server 类实现

在 RPC 类实现小节中我们介绍了 RPC 服务器端代码获取 Server 对象的流程，服务器端代码获取了 Server 对象后，会启动这个 Server 对象监听网络上的 RPC 请求并触发响应操作。本节就介绍 Server 类的具体实现。

为了提高性能，Server 类采用了很多技术来提高并发能力，包括线程池、JavaNIO 提供的 Reactor 模式等，其中 Reactor 模式贯穿了整个 Server 的设计。

1. Reactor 模式

RPC 服务器端代码的处理流程与所有网络程序服务器端的处理流程类似，都分为 5 个步骤：①读取请求；②反序列化请求；③处理请求；④序列化响应；⑤发回响应。对于网络服务器端程序来说，如果对每个请求都构造一个线程响应，那么在负载增加时性能会下降得很快；而如果只用少量线程响应，又会在 IO 阻塞时造成响应流程停滞、吞吐率降低。所以为了解决上述问题，Reactor 模式出现了。

Reactor 模式是一种广泛应用在服务器端的设计模式，也是一种基于事件驱动的设计模式。Reactor 模式的处理流程是：应用程序向一个中间人注册 IO 事件，当中间人监听到这个 IO 事件发生后，会通知并唤醒应用程序处理这个事件。这里的中间人其实是一个不断等待和循环的线程，它接受所有应用程序的注册，并检查应用程序注册的 IO 事件是否就绪，如果就绪了则通知应用程序进行处理。

一个简单的基于 Reactor 模式的网络服务器设计如图 2-16 所示，包括 reactor、acceptor、以及 handler 等模块。reactor 负责监听所有的 IO 事件，当检测到一个新的 IO 事件发生时，reactor 会唤醒这个事件对应的模块处理。acceptor 则负责响应 Socket 连接请求事件，acceptor 会接收请求建立连接，之后构造 handler 对象。handler 对象则负责向 reactor 注册 IO 读事件，然后从网络上读取请求并执行对应的业务逻辑，最后发回响应。使用 Reactor 模式的服务器响应客户端请求的流程可以分为如下几个步骤。

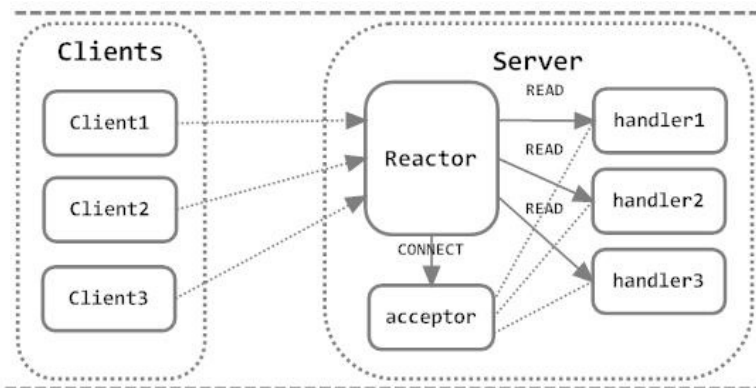


图 2-16 基于 Reactor 模式的网络服务器设计

- 客户端发送 Socket 连接请求到服务器，服务器端的 reactor 对象监听到了这个 IO 请求，由于 acceptor 对象在 reactor 对象上注册了 Socket 连接请求的 IO 事件，所以 reactor 会触发 acceptor 对象响应 Socket 连接请求。
- acceptor 对象会接收来自客户端的 Socket 连接请求，并为这个连接创建一个 handler 对象，handler 对象的构造方法会在 reactor 对象上注册 IO 读事件。
- 客户端在连接建立成功之后，会通过 Socket 发送 RPC 请求。RPC 请求到达 reactor 后，会由 reactor 对象分发（dispatch）到对应的 handler 对象处理。
- handler 对象会从网络上读取 RPC 请求，然后反序列化请求并执行对应的逻辑，最后将响应消息序列化并通过 Socket 发回客户端。至此，一个完整的 RPC 请求流程就结束了。

采用了基于事件驱动模式的 Reactor 结构，服务器只有在指定的 IO 事件发生时才会调用 acceptor 以及 handler 对象提供的方法执行业务逻辑，避免了在 IO 上无谓的阻塞，也就提高了服务器的效率。但是由于上述设计中服务器端只有一个线程，所以就要求 handler 中读取请求、执行请求以及发送响应的流程必须能够迅速处理完，如果在任意一个环节阻塞了，则整个服务器逻辑全部阻塞。所以需要进一步改进架构，也就是使用多线程处理业务逻辑。

对于 handler 处理 RPC 请求的 5 个步骤，我们可以将占用时长较长的读取请求部分以及业务逻辑处理部分交给两个独立的线程池处理。图 2-17 给出了使用多线程的 Reactor 模式的网络服务器结构，readers 线程池中包含若干个执行读取 RPC 请求任务的 Reader 线程，它们会在 Reactor 上注册读取 RPC 请求的 IO 事件，然后从网络中读取 RPC 请求，并将 RPC 请求封装在一个 Call 对象中，最后将 Call 对象放入共享消息队列 MQ 中。而 handlers 线程池则包含若干个处理业务逻辑的 Handler 线程，它们会不断地从共享消息队列 MQ 中取出 RPC 请求，然后执行业务逻辑并向客户端发回响应。这种结构保证了 IO 事件的监听和分发，RPC 请求的读取和响应是在不同的线程中执行的，大大提高了服务器的并发性能。

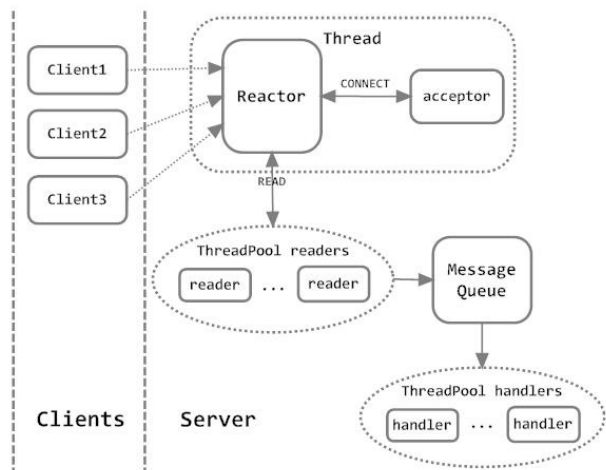


图 2-17 基于多线程 Reactor 模式的网络服务器结构

采用了多线程的 Reactor 模式后，IO 事件的监听、RPC 请求的读取和处理就可以并发地进行了，但是对于像 Namenode 这种分布式集群中的 Master 节点来说，同一时间可能有非常多的 Socket 连接请求以及 RPC 请求到达，这就可能造成 Reactor 在处理 and 分发这些 IO 事件时出现拥塞，导致服务器整体性能降低。所以我们可以将图 2-17 中的一个 Reactor 对象扩展成多个 Reactor，它们分别用于并发地监听不同的 IO 事件，这样也就提高了 IO 事件的处理效率，同时提高了 RPC 服务器的性能。

图 2-18 给出了使用多个 Reactor 的服务器结构，这里的 `mainReactor` 负责监听 Socket 连接事件，`readReactor` 负责监听 IO 读事件，`respondSelector` 负责监听 IO 写事件。由于同一时间到达 RPC 服务器的 RPC 请求可能很多，也就会造成一个 `readReactor` 要同时处理多个 IO 读事件的分发，当系统负载达到一定量时，`readReactor` 就有可能成为瓶颈。所以我们可以构造多个 `readReactor` 对象，不同的 Reader 线程会根据一定的逻辑到不同的 `readReactor` 上注册 IO 读事件。当 `acceptor` 建立了 Socket 连接后，会从 `readers` 线程池中取出一个 Reader 线程触发读取 RPC 请求的流程。Reader 线程会根据一定的逻辑选出一个 `readReactor` 对象并在这个 `readReactor` 对象上注册读取 RPC 请求的 IO 事件，之后就会由该 `readReactor` 在网络上监听是否有 RPC 请求到达，并触发 Reader 线程读取流程了。当 Handler 成功地处理了一个 RPC 请求后，它会向 `respondSelector` 注册写 RPC 响应 IO 事件，当 Socket 输出流管道可以写数据时，`Sender` 类就可以将响应信息发回客户端了。

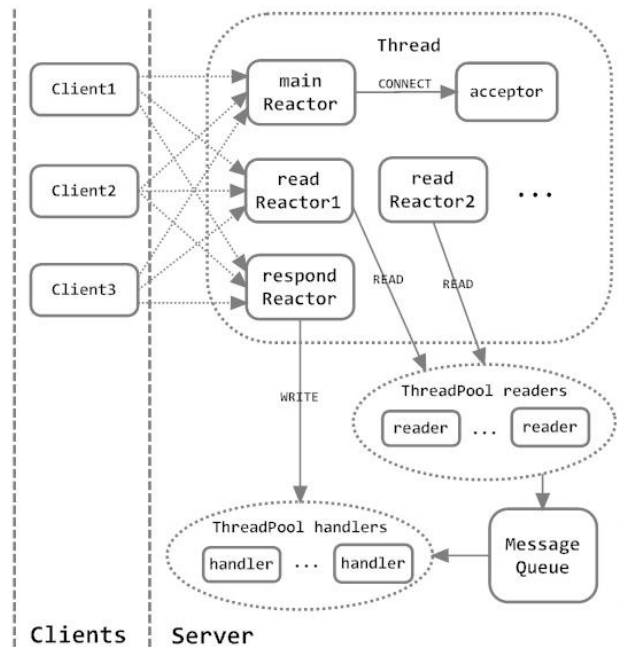


图 2-18 基于多 Reactor 多线程模式的网络服务器结构

Server 类正是采用了多线程加多 Reactor 的设计,我们在下一小节中介绍 Server 类的结构。

2. Server 类设计

Server 类的设计结构基本与图 2-18 类似，是一个典型的多线程加多 Reactor 的网络服务器结构。Server 定义了如下几个内部类，我们可以对比 Reactor 结构中的模块来理解。

- **Listener**: 类似于 Reactor 模式中的 mainReactor。Listener 对象中存在一个 Selector 对象 acceptSelector，负责监听来自客户端的 Socket 连接请求。当 acceptSelector 监听到连接请求后，Listener 对象会初始化这个连接，之后采用轮询的方式从 readers 线程池中选出一个 Reader 线程处理 RPC 请求的读取操作。
- **Reader**: 与 Reactor 模式中的 Reader 线程相同，用于读取 RPC 请求。Reader 线程类中存在一个 Selector 对象 readSelector，类似于 Reactor 模式中的 readReactor，这个对象用于监听网络中是否有可以读取的 RPC 请求。当 readSelector 监听到有可读的 RPC 请求后，会唤醒 Reader 线程读取这个请求，并将请求封装在一个 Call 对象中，然后将这个 Call 对象放入共享队列 CallQueue 中。
- **Handler**: 与 Reactor 模式中的 Handler 类似，用于处理 RPC 请求并发回响应。Handler 对象会从 CallQueue 中不停地取出 RPC 请求，然后执行 RPC 请求对应的本地函数，最后封装响应并将响应发回客户端。为了能够并发地处理 RPC 请求，Server 中会存在多个 Handler 对象。
- **Responder**: 用于向客户端发送 RPC 响应，读者可能会问，在 Handler 中不是已经发送 RPC 响应了吗？为什么还需要再实现一个 Responder 类？这是因为，在响应很大或者网络条件不佳等情况下，Handler 线程很难将完整的响应发回客户端，这就会造成 Handler 线程阻塞，从而影响 RPC 请求的处理效率。所以 Handler 在没能够将完整的 RPC 响应发回客户端时，会在 Responder 内部的 respondSelector 上注册一个写响应事件，这里的 respondSelector 与 Reactor 模式的 respondSelector 概念相同，当 respondSelector 监听到网络情况具备写响应的条件时，会通知 Responder 将剩余响应发回客户端。

了解了 Server 类的设计后，我们再来学习 Server 处理客户端 RPC 请求的流程。如图 2-19 所示，Server 类处理 RPC 请求的流程可以分为如下几个步骤。

- Listener 线程的 acceptSelector 在 ServerSocketChannel 上注册 OP_ACCEPT 事件，并且创建 readers 线程池。每个 Reader 的 readSelector 此时并不监听任何 Channel。
- Client 发送 Socket 连接请求，触发 Listener 的 acceptSelector 唤醒 Listener 线程。
- Listener 调用 ServerSocketChannel.accept() 创建一个新的 SocketChannel。
- Listener 从 readers 线程池中挑选一个线程，并在 Reader 的 readSelector 上注册 OP_READ 事件。
- Client 发送 RPC 请求数据包，触发 Reader 的 selector 唤醒 Reader 线程。
- Reader 从 SocketChannel 中读取数据，封装成 Call 对象，然后放入共享队列 CallQueue 中。
- 最初，handlers 线程池中的线程都在 CallQueue（调用 BlockingQueue.take()）上阻塞，当有 Call 对象被放入后，其中一个 Handler 线程被唤醒，然后根据 Call 对象的信息

调用 `BlockingService` 对象的 `callBlockingMethod()` 方法。随后, `Handler` 尝试将响应写入 `SocketChannel`。

- 如果 `Handler` 发现无法将响应完全写入 `SocketChannel`, 将在 `Responder` 的 `respondSelector` 上注册 `OP_WRITE` 事件。当 `Socket` 恢复正常时, `Responder` 将被唤醒, 继续写响应。当然, 如果一个 `Call` 响应在一定时间内都无法被写入, 则会被 `Responder` 移除。

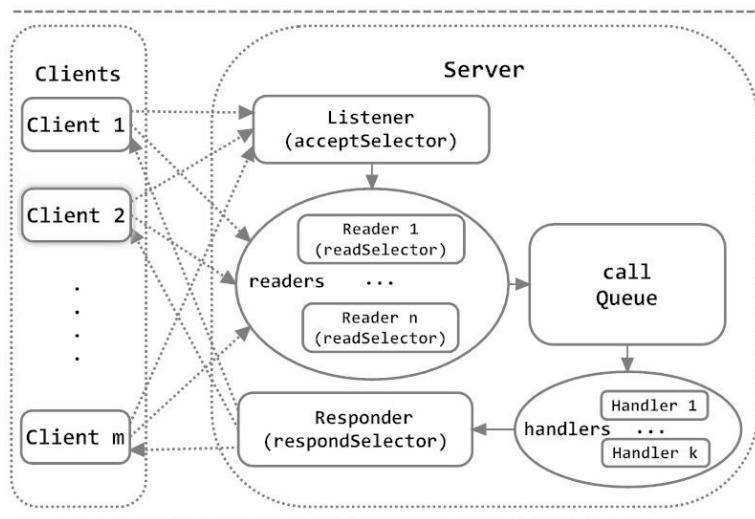


图 2-19 Server 处理客户端 RPC 请求流程图

了解了 `Server` 类的设计以及处理 RPC 请求的流程后, 再来学习 `Server` 类的代码实现就比较简单了, 下面就介绍 `Server` 类的具体实现。

3. Server 类实现

在上一节中我们介绍了 `Server` 类的设计, 包括 `Listener`、`Reader`、`Handler`、`Responder` 类的作用以及相互之间的关系, 本节则从代码角度来分析这几个类的实现。

(1) 内部类 `Listener`

`Listener` 是一个线程类, 整个 `Server` 中只会有一个 `Listener` 线程, 用于监听来自客户端的 `Socket` 连接请求。对于每一个新到达的 `Socket` 连接请求, `Listener` 都会从 `readers` 线程池中选择 一个 `Reader` 线程来处理。

`Listener` 类中定义了一个 `Selector` 对象, 负责监听 `SelectionKey.OP_ACCEPT` 事件, `Listener` 线程的 `run()` 方法会循环判断是否监听到了 `OP_ACCEPT` 事件, 也就是是否有新的 `Socket` 连接请求到达, 如果有则调用 `doAccept()` 方法响应。 `Listener.run()` 方法的代码如下:

```
public void run() {
    while (running) {
```

```

SelectionKey key = null;
try {
    getSelector().select();
    // 循环判断是否有新的连接建立请求
    Iterator<SelectionKey> iter = getSelector().selectedKeys().iterator();
    while (iter.hasNext()) {
        key = iter.next();
        iter.remove();
        try {
            if (key.isValid()) {
                if (key.isAcceptable())
                    // 如果有, 则调用 doAccept() 方法响应
                    doAccept(key);
            }
        } catch (IOException e) {
        }
        key = null;
    }
} catch (OutOfMemoryError e) {
    // 这里可能出现内存溢出的情况, 要特别注意
    closeCurrentConnection(key, e);
    cleanupConnections(true);
    try { Thread.sleep(60000); } catch (Exception ie) {}
} catch (Exception e) {
    // 捕获到其他异常, 也关闭当前连接
    closeCurrentConnection(key, e);
}
cleanupConnections(false);
}

// running == false 时, 关闭 Listener 线程
// ...
}

```

下面看一下 `doAccept()` 方法的实现。`doAccept()` 方法会接收来自客户端的 `Socket` 连接请求并初始化 `Socket` 连接。之后 `doAccept()` 方法会从 `readers` 线程池中选出一个 `Reader` 线程读取来自这个客户端的 `RPC` 请求。每个 `Reader` 线程都会有一个自己的 `readSelector`, 用于监听是否有新的 `RPC` 请求到达。所以 `doAccept()` 方法在建立连接并选出 `Reader` 对象后, 会在这个 `Reader` 对象的 `readSelector` 上注册 `OP_READ` 事件。那么这里就有一个问题了, `Reader` 对象在被通知时是怎么知道从哪个 `Socket` 输入流上读取数据呢? 这里就用到了 `Connection` 类, `Connection` 类封装了 `Server` 与 `Client` 之间的 `Socket` 连接, `doAccept()` 方法会通过 `SelectionKey` 将新构造的 `Connection` 对象传给 `Reader`, 这样 `Reader` 线程在被唤醒时就可以通过 `Connection` 对象读取 `RPC` 请求了。`doAccept()` 方法的代码如下:

```

void doAccept(SelectionKey key) throws IOException, OutOfMemoryError {
    // 接收请求, 建立连接
    Connection c = null;

```

Hadoop 2.X HDFS 源码剖析

```
ServerSocketChannel server = (ServerSocketChannel) key.channel();
SocketChannel channel;
while ((channel = server.accept()) != null) {
    channel.configureBlocking(false);
    channel.socket().setTcpNoDelay(tcpNoDelay);

    // 从 readers 线程池中取出一个 Reader 线程
    Reader reader = getReader();
    try {
        // 唤醒处于等待状态的 readSelector
        reader.startAdd();
        // 注册 IO 读事件
        SelectionKey readKey = reader.registerChannel(channel);
        // 构造 Connection 对象, 添加到 readKey 的附件传递给 Reader 对象
        c = new Connection(readKey, channel, Time.now());
        readKey.attach(c);
        synchronized (connectionList) {
            connectionList.add(numConnections, c);
            numConnections++;
        }
    } finally {
        reader.finishAdd();
    }
}
```

这里的 Reader 其实就是一个独立的线程, 专门负责读操作。我们接下来看 Reader 类的实现。

(2) 内部类 Reader (done)

Reader 也是一个线程类, 每个 Reader 线程都会负责读取若干个客户端连接发来的 RPC 请求。而在 Server 类中会存在多个 Reader 线程构成一个 readers 线程池, readers 线程池并发地读取 RPC 请求, 提高了 Server 处理 RPC 请求的速率。Reader 类定义了自己的 readSelector 字段, 用于监听 SelectionKey.OP_READ 事件。Reader 类还定义了 adding 字段标识是否有任务正在添加到 Reader 线程。

```
private volatile boolean adding = false;
private final Selector readSelector;
```

Reader 线程的主循环则是在 doRunLoop()方法中实现的, doRunLoop()方法会监听当前 Reader 对象负责的所有客户端连接中是否有新的 RPC 请求到达, 如果有则读取这些请求, 然后将成功读取的请求用一个 Call 对象封装, 最后放入 callQueue 中等待 Handler 线程处理。doRunLoop()方法的代码如下:

```
private synchronized void doRunLoop() {
    while (running) {
        SelectionKey key = null;
```



```

try {
    readSelector.select();
    // 有任务添加时等待；在任务添加完成之后会被唤醒
    while (adding) {
        this.wait(1000);
    }

    // 在当前的 readSelector 上等待可读事件，也就是有客户端 RPC 请求到达
    Iterator<SelectionKey> iter = readSelector.selectedKeys().iterator();
    while (iter.hasNext()) {
        key = iter.next();
        iter.remove();
        if (key.isValid()) {
            if (key.isReadable()) {
                // 有可读事件时，调用 doRead() 方法处理
                doRead(key);
            }
        }
        key = null;
    }
} catch (InterruptedException e) {
    if (running) { // 出现异常，则记录在日志中
        LOG.info(getName() + " unexpectedly interrupted", e);
    }
} catch (IOException ex) {
    LOG.error("Error in Reader", ex);
}
}
}

```

`doRead()`方法负责读取 RPC 请求，虽然 `readSelector` 监听到了 RPC 请求的可读事件，但是 `doRead()`方法此时并不知道这个 RPC 请求是由哪个客户端发送来的，所以 `doRead()`方法首先会调用 `SelectionKey.attachment()`方法获取 `Listener` 对象构造的 `Connection` 对象，`Connection` 对象中封装了 `Server` 与 `Client` 之间的网络连接，之后 `doRead()`方法只需调用 `Connection.readAndProcess()`方法就可以读取 RPC 请求了，这里的设计非常的巧妙。

```

void doRead(SelectionKey key) throws InterruptedException {
    int count = 0;
    // 通过 SelectionKey 获取 Connection 对象
    Connection c = (Connection)key.attachment();
    if (c == null) {
        return;
    }
    c.setLastContact(Time.now());

    try {
        count = c.readAndProcess(); // 调用 Connection.readAndProcess 处理读取请求
    } catch (InterruptedException ieo) {
    }
}

```

```
        throw ieo;
    } catch (Exception e) {
        count = -1;
    }
    if (count < 0) {
        closeConnection(c);
        c = null;
    }
    else {
        c.setLastContact(Time.now());
    }
}
```

下面我们看一下 Connection 类的实现。

(3) 内部类 Connection (done)

Connection 类维护了 Server 与 Client 之间的 Socket 连接。Reader 线程会调用 readAndProcess()方法从 IO 流中读取一个 RPC 请求。

readAndProcess()方法会首先从 Socket 流中读取连接头域 (connectionHeader)，然后读取一个完整的 RPC 请求，最后调用 processOneRpc()方法处理这个 RPC 请求。processOneRpc()方法会读取出 RPC 请求头域，然后调用 processRpcRequest()处理 RPC 请求体。这里特别注意，如果在处理过程中抛出了异常，则直接通过 Socket 返回 RPC 响应 (带有 Server 异常信息的响应)。processOneRpc()方法的代码如下：

```
private void processOneRpc(byte[] buf)
    throws IOException, WrappedRpcServerException, InterruptedException {
    int callId = -1;
    int retry = RpcConstants.INVALID_RETRY_COUNT;
    try {
        final DataInputStream dis =
            new DataInputStream(new ByteArrayInputStream(buf));
        // 解析出 RPC 请求头域
        final RpcRequestHeaderProto header =
            decodeProtobufFromStream(RpcRequestHeaderProto.newBuilder(), dis);
        callId = header.getCallId(); // 从 RPC 请求头域中提取出 callId
        retry = header.getRetryCount(); // 从 RPC 请求头域中提取出重试次数
        checkRpcHeaders(header);

        // 处理 RPC 请求头域异常的情况
        if (callId < 0) { during connection setup
            processRpcOutOfBandRequest(header, dis);
        } else if (!connectionContextRead) {
            throw new WrappedRpcServerException(
                RpcErrorCodeProto.FATAL_INVALID_RPC_HEADER,
                "Connection context not established");
        } else {
            // 如果 RPC 请求头域正常，则直接调用 processRpcRequest 处理 RPC 请求体
        }
    }
}
```

```

        processRpcRequest(header, dis);
    }
} catch (WrappedRpcServerException wrse) { // 直接发回异常, 通知 Client
    Throwable ioe = wrse.getCause();
    final Call call = new Call(callId, retry, null, this);
    setupResponse(authFailedResponse, call,
        RpcStatusProto.FATAL, wrse.getRpcErrorCodeProto(), null,
        ioe.getClass().getName(), ioe.getMessage());
    // 通过 Socket 返回这个带有异常信息的 RPC 响应
    responder.doRespond(call);
    throw wrse;
}
}
}

```

对于一个正常的 RPC 请求, `processOneRpc()` 方法会调用 `processRpcRequest()` 方法处理。`processRpcRequest()` 会从输入流中解析出完整的请求对象 (包括请求元数据以及请求参数), 然后根据 RPC 请求头的信息 (包括 `callId`) 构造 `Call` 对象 (`Call` 对象保存了这次调用的所有信息), 最后将这个 `Call` 对象放入 `callQueue` 队列中保存, 等待 `Handler` 线程处理。

```

private void processRpcRequest(RpcRequestHeaderProto header,
    DataInputStream dis) throws WrappedRpcServerException,
    InterruptedException {

    // ...
    Writable rpcRequest;
    try { // 读取 RPC 请求体
        rpcRequest = ReflectionUtils.newInstance(rpcRequestClass, conf);
        rpcRequest.readFields(dis);
    } catch (Throwable t) { // includes runtime exception from newInstance
        // 出现异常则直接抛出, 在上一层捕获异常
        throw new WrappedRpcServerException(
            RpcErrorCodeProto.FATAL_DESERIALIZING_REQUEST, err);
    }

    // 构造 Call 对象封装 RPC 请求信息
    Call call = new Call(header.getCallId(), header.getRetryCount(),
        rpcRequest, this, ProtoUtil.convert(header.getRpcKind()), header
            .getClientId().toByteArray());
    // 将 Call 对象放入 callQueue 中, 等待 Handler 处理
    callQueue.put(call);
    incRpcCount();
}
}

```

下面我们就学习 `Handler` 线程是如何处理这个 `Call` 对象的。

(4) 内部类 `Handler` (done)

`Handler` 类也是一个线程类, 负责执行 RPC 请求对应的本地函数, 然后将结果发回客户端。在 `Server` 类中会有多个 `Handler` 线程, 它们并发地处理 RPC 请求。

Handler 线程类的主方法会循环从共享队列 `callQueue` 中取出待处理的 `Call` 对象，然后调用 `Server.call()` 方法执行 RPC 调用对应的本地函数，如果在调用过程中发生异常，则将异常信息保存下来。接下来 `Handler` 会调用 `setupResponse()` 方法构造 RPC 响应，并调用 `responder.doRespond()` 方法将响应发回。

```
while (running) {
    try {
        // ...
        // 从 callQueue 中取出请求
        final Call call = callQueue.take();
        try {
            if (call.connection.user == null) {
                // 通过 call() 发起本地调用，并返回结果
                value = call(call.rpcKind, call.connection.protocolName, call.rpcRequest,
                    call.timestamp);
            }
            // ...
        } catch (Throwable e) {
            // ...
            // 如果在调用过程中发生异常，则将异常信息保存下来
            if (e instanceof RpcServerException) {
                RpcServerException rse = ((RpcServerException)e);
                returnStatus = rse.getRpcStatusProto();
                detailedErr = rse.getRpcErrorCodeProto();
            } else {
                returnStatus = RpcStatusProto.ERROR;
                detailedErr = RpcErrorCodeProto.ERROR_APPLICATION;
            }
            errorClass = e.getClass().getName();
            error = StringUtils.stringifyException(e);
            String exceptionHdr = errorClass + ": ";
            if (error.startsWith(exceptionHdr)) {
                error = error.substring(exceptionHdr.length());
            }
        }
        CurCall.set(null);
        synchronized (call.connection.responseQueue) {
            // 构造 RPC 响应，如果调用正常就返回结果，有异常则返回异常信息
            setupResponse(buf, call, returnStatus, detailedErr,
                value, errorClass, error);

            // 调用 responder.doRespond() 返回响应
            responder.doRespond(call);
        }
    } catch (InterruptedException e) {
        // ...
    } catch (Exception e) {
        // ...
    }
}
```

`Server.call()`方法执行本地函数的部分我们在服务器获取 `Server` 对象小节中已经介绍过了，这里不再重复介绍。

(5) 内部类 `Responder` (done)

内部类 `Responder` 也是一个线程类，`Server` 端仅有一个 `Responder` 对象，`Responder` 内部包含一个 `Selector` 对象 `responseSelector`，用于监听 `SelectionKey.OP_WRITE` 事件。当网络环境不佳或者响应信息太大时，`Handler` 线程可能无法发送完整的响应信息到客户端，这时 `Handler` 会在 `Responder.responseSelector` 上注册 `SelectionKey.OP_WRITE` 事件，`responseSelector` 会循环监听网络环境是否具备发送数据的条件，之后 `responseSelector` 会触发 `Responder` 线程发送未完成的响应结果到客户端。由于篇幅原因，这里不再贴出 `Responder` 类的代码实现，感兴趣的读者可以自行阅读。

第 3 章 Namenode（名字节点）

HDFS 集群是以 Master/Slave 模式运行的，主要有两类节点：Namenode（名字节点）和 Datanode（数据节点）。Namenode 是 HDFS 中的主节点，本章介绍 Namenode 的实现。

对于 Namenode 的分析，我们分为以下几个部分来介绍。

- 文件系统目录树管理：HDFS 的目录和文件在内存中是以一棵树的形式存储的，这个目录树结构是由 Namenode 维护的，Namenode 会修改这个树形结构以对外提供添加和删除文件等操作功能。文件系统目录树上的节点还保存了 HDFS 文件与数据块的对应关系，我们知道 HDFS 中的每个文件都是被拆分成若干数据块冗余存放的，文件与数据块的对应关系也是由 Namenode 维护的。
- 数据块以及数据节点管理：HDFS 中的数据块是冗余备份在集群中的数据节点上的，所以 Namenode 还需要维护数据块与数据节点之间的对应关系。这里的对应关系包括两个部分：①数据块存放在哪些数据节点上；②一个数据节点上保存了哪些数据块。
- 租约管理：租约是 Namenode 给予租约持有者（LeaseHolder，一般是 HDFS 客户端）在规定时间内拥有文件权限（写文件）的合同，Namenode 会执行租约的发放、回收、检查以及恢复等操作。
- 缓存管理：Hadoop 2.3.0 版本新增了集中式缓存管理功能（Centralized Cache Management），允许用户将一些文件和目录保存到 HDFS 缓存中。HDFS 的集中式缓存是由分布在 Datanode 上的堆外内存组成的，并且由 Namenode 统一管理。
- FSNamesystem：Namenode 涉及很多 HDFS 的处理逻辑，例如读文件、写文件、追加写文件等，Namenode 的 FSNamesystem 类是管理这些逻辑的门面类，也是 Namenode 中最重要的一个类。
- Namenode 的启动和停止：Hadoop 2.X 实现中提供了 HA 功能，HA 集群中会存在两种状态的 Namenode：Active Namenode 作为服务节点，Standby Namenode 作为热备节点。Namenode 在启动时会先进入安全模式，在安全模式中的 Namenode 不会接受客户端对命名空间的修改，Namenode 成功启动后会离开安全模式进入 Standby 状态。

下面我们分别介绍上述几个部分在 Namenode 中的实现。

3.1 文件系统目录树

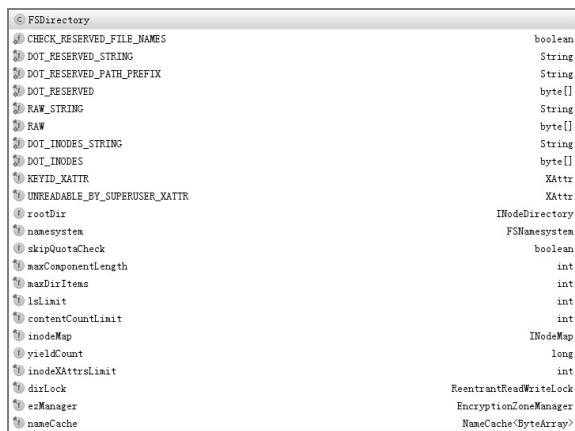
Namenode 最重要的两个功能之一就是维护文件系统的命名空间（namesystem）。HDFS

文件系统的命名空间（namespace）是以“/”为根的整体目录树，是通过 FSDirectory 类来管理的。

HDFS 文件系统的命名空间在 Namenode 的内存中是以一颗树的结构来存储的。在 HDFS 中，不管是目录还是文件，在文件系统目录树中都被看作是一个 INode 节点。如果是目录，则其对应的类为 INodeDirectory；如果是文件，则其对应的类为 INodeFile。INodeDirectory 以及 INodeFile 类都是 INode 的派生类。INodeDirectory 中包含一个成员集合变量 children，如果该目录下有子目录或者文件，其子目录或文件的 INode 引用就会被保存在 children 集合中。HDFS 就是通过这种方式来维护整个文件系统的目录结构的。

HDFS 会将命名空间保存到 Namenode 的本地文件系统上一个叫 fsimage（命名空间镜像）的文件中。利用这个文件，Namenode 每次重启时都能将整个 HDFS 的命名空间重构，fsimage 文件的操作由 FSImage 类负责。另外，对 HDFS 的各种操作，Namenode 都会在操作日志（editlog）中进行记录，以便周期性地将该日志与 fsimage 进行合并生成新的 fsimage。该日志文件也在 Namenode 的本地文件系统中保存，叫 editlog 文件，editlog 的相关操作由 FSEditLog 类管理。

如图 3-1 所示，FSDirectory 维护着文件系统目录树的根节点（这个根节点是整个文件系统的 root，是一个 INodeDirectory 类型），本章就以 INode 作为分析入口。



Field Name	Type
CHECK_RESERVED_FILE_NAMES	boolean
DOT_RESERVED_STRING	String
DOT_RESERVED_PATH_PREFIX	String
DOT_RESERVED	byte[]
RAW_STRING	String
RAW	byte[]
DOT_INODES_STRING	String
DOT_INODES	byte[]
KEYID_XATTR	XAttr
UNREADABLE_BY_SUPERUSER_XATTR	XAttr
rootDir	INodeDirectory
namesystem	FSNamesystem
skipQuotaCheck	boolean
maxComponentLength	int
maxDirItems	int
lsLimit	int
contentCountLimit	int
inodeMap	INodeMap
yieldCount	long
inodeXAttrLimit	int
dirLock	ReentrantReadWriteLock
exManager	EncryptionZoneManager
nameCache	NameCache<ByteArray>

图 3-1 FSDirectory 字段列表

3.1.1 INode 相关类

看到 INode，我们可以马上联想到 Linux 的 inode，即索引节点。索引节点保存了 Linux 文件的元信息，如文件类型与权限、所有者标识和以字节为单位的文件长度等。在索引节点的后半部分，则存放着数据块索引，也就是文件或目录数据在磁盘上的位置。

HDFS 就是借鉴了 Linux 的 inode，将 HDFS 中文件和目录的抽象类命名为 INode。在 HDFS 的文件系统目录树中，不管是目录还是文件，都会被看作是一个 INode 类的引用。INode 类

是一个抽象类，是 `InodeDirectory` 和 `InodeFile` 的父类。如果是目录，则其实际类为 `InodeDirectory`；如果是文件，则其对应的类为 `InodeFile`。`InodeDirectory` 中包含一个成员集合变量 `children`，如果该目录下有子目录或者文件，其子目录或文件的引用就会保存在 `children` 集合中。

HDFS 中 `Inode` 继承关系图如图 3-2 所示，我们依次介绍 `Inode` 相关的所有类。

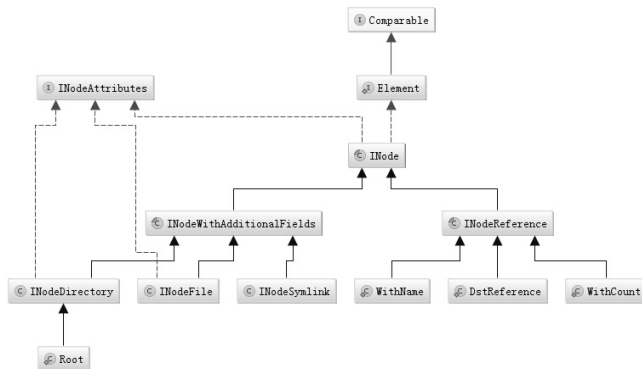


图 3-2 Inode 继承关系图

1. Inode 抽象类

`Inode` 类是整个 `Inode` 体系的根接口，它是一个抽象类，保存了 HDFS 目录和文件的所有共同属性，包括当前节点的父节点的 `Inode` 对象的引用（只能是 `InodeDirectory` 类或者 `InodeReference` 类）、文件/目录名、用户组、访问权限、最后修改时间、上次访问时间、完整路径名、文件扩展属性等。

(1) Inode 类方法

如图 3-2 所示，`Inode` 类实现了 `InodeAttributes` 接口。这个接口包括以下 7 个字段的 `get` 方法。

- `userName`：文件/目录所属用户名。
- `groupName`：文件/目录所属组名。
- `fsPermission`：文件/目录访问权限。
- `aclFeature`：安全相关。
- `modificationTime`：文件/目录上次修改时间。
- `accessTime`：文件/目录上次访问时间。
- `XAttrFeature`：当前文件/目录的扩展属性（`ExtendedAttributes`）。

文件系统扩展属性是目前流行的 POSIX 系统中文件系统具有的一项特殊功能，可以给文件、文件夹添加额外的 `key/value` 键值对，键和值都是字符串并且有一定长度的限制。文件系统扩展属性使得现有的文件系统得以支持在原始设计中未提供的功能。

Inode 抽象类除了实现 InodeAttributes 接口中的方法外, 还定义了 Inode 元信息的 get 与 set 接口方法。Inode 元信息包括如下一些信息。

- id: Inode 的 id。
- name: 文件/目录的名称。
- fullPathName: 文件/目录的完整路径。
- parent: 文件/目录的父节点。

同时 Inode 还提供了如下几个基本的判断方法。

- isFile(): 判断是否为文件。
- isDirectory(): 判断是否为目录。
- isSymlink(): 判断是否为符号链接。
- isRoot(): 判断是否为文件系统目录树的根节点。

这里需要注意的是, Inode 类的设计采用了模板模式。Inode 类定义的方法多为两个, 其中一个为 final 的接口方法, 用于规范接口的调用; 另一个则是 abstract 的抽象方法, 抽象方法留给子类具体实现。Inode 类实现的 InodeAttributes 接口中定义的方法就是采用了这种模式, 将 userName 等字段的定义留给子类实现, 这里我们以 setUser() 方法为例。

```
// 抽象方法, 具体实现留给子类
abstract void setUser(String user);

// 模板方法, 是 final 的, 不可以继承, 供接口调用
final Inode setUser(String user, int latestSnapshotId)
    throws QuotaExceededException {
    recordModification(latestSnapshotId);
    setUser(user);
    return this;
}
```

(2) Inode 类字段

Inode 类中只有一个字段, 就是 parent, 表明当前 Inode 的父目录。HDFS 中除了根目录外, 其他所有的文件与目录都存在一个父目录。注意, 父目录的类型只能是 InodeDirectory 类或者 InodeReference 类之一。

```
private Inode parent = null;
```

2. InodeWithAdditionalFields 类

上一节介绍了 Inode 这个根接口类, 但是 Inode 类只定义了一个字段 parent, 其余字段的值都是通过抽象的 get() 方法获得的, 并且留给了子类来定义。如图 3-3 所示, InodeWithAdditionalFields 类就定义了这些字段——id、name、permission、modificationTime、accessTime 等, 并且覆盖了 Inode 中对应的抽象方法。

InodeWithAdditionalFields	
① id	long
② name	byte[]
③ permission	long
④ modificationTime	long
⑤ accessTime	long
⑥ next	LinkedElement
⑦ EMPTY_FEATURE	Feature[]
⑧ features	Feature[]

图 3-3 InodeWithAdditionalFields 类字段

这里比较有意思的是 permission 字段和 features 字段, 我们重点讲一下这两个字段。

(1) permission 字段

permission 字段主要包括 3 个部分的信息：用户信息、用户组信息和权限信息。

permission 字段是 long 类型的，其中前 16 个比特用来存放文件模式标识（mode，类似于 Linux 中的 777），中间 25 个比特用来存放用户组标识（group），最后 23 个比特用来存放用户名标识（user）。枚举类 PermissionStatusFormat 就是用来解析以及处理 permission 字段的工具类，提供了获取 permission 字段中 mode、group、user 部分对应的文件模式、用户组名以及用户名的方法。在 HDFS 中，用户名和用户标识的对应关系、用户组名和用户组标识的对应关系都保存在 SerialNumberManager 类中。通过 SerialNumberManager 类，名字节点不必在 INode 对象中保存字符串形式的用户名和用户组名，只需将整型的用户名标识和用户组名标识放入 permission 字段中即可。这种设计节省了 INode 对象对内存的占用，是一个非常巧妙的优化。

PermissionStatusFormat 类的代码如下所示，可以看到 PermissionStatusFormat 枚举类使用到了 LongBitFormat 类来存储与操作底层的 permission 字段。LongBitFormat 的 combine() 方法和 retrieve() 方法主要是对底层的 permission 字段进行掩码操作、移位操作以及逻辑与操作，比较简单，这里就不详细叙述了，感兴趣的读者可以自己学习。

```
static enum PermissionStatusFormat {
    MODE(null, 16),
    GROUP(MODE.BITS, 25),
    USER(GROUP.BITS, 23);

    final LongBitFormat BITS;

    private PermissionStatusFormat(LongBitFormat previous, int length) {
        BITS = new LongBitFormat(name(), previous, length, 0);
    }

    // 提取最后 23 个比特的 user 信息，并通过 SerialNumberManager 获取用户名
    static String getUser(long permission) {
        // 首先获取 permission 中最后 23 个比特的用户标识
        final int n = (int)USER.BITS.retrieve(permission);
        // 通过 SerialNumberManager 类获取用户标识对应的用户名
        return SerialNumberManager.INSTANCE.getUser(n);
    }

    // 提取中间 25 个比特的 group 信息，并通过 SerialNumberManager 获取用户组名
    static String getGroup(long permission) {
        // 首先获取 permission 中间 25 个比特的用户组标识
        final int n = (int)GROUP.BITS.retrieve(permission);
        // 使用 SerialNumberManager 获取用户组标识对应的用户组名
        return SerialNumberManager.INSTANCE.getGroup(n);
    }

    // 提取前 16 个比特的 mode 信息
    static short getMode(long permission) {
```

```

        return (short)MODE.BITS.retrieve(permission);
    }

    // 将一个 PermissionStatus 类, 转换成 long 类型的 permission 信息
    static long toLong(PermissionStatus ps) {
        long permission = 0L;
        final int user = SerialNumberManager.INSTANCE.getUserSerialNumber(
            ps.getUserName());
        permission = USER.BITS.combine(user, permission);
        final int group = SerialNumberManager.INSTANCE.getGroupSerialNumber(
            ps.getGroupName());
        permission = GROUP.BITS.combine(group, permission);
        final int mode = ps.getPermission().toShort();
        permission = MODE.BITS.combine(mode, permission);
        return permission;
    }
}

```

(2) features 字段

在 HDFS 2.2 版本前, INode 的子类还包括 InodeDirectoryWithQuota 和 InodeFileUnderConstrution, 这两个类分别用于描述带有磁盘配额的目录以及正在进行写操作的文件。但在 HDFS 2.6 版本中, 这两个子类从 INode 的类图中消失了。这是因为 HDFS 2.6 版本的代码将磁盘配额、正在构建(UnderConstrution)、快照(Snapshot)等功能抽象成 INode 的特性(Feature), 可以将特性添加到 INode 上, 使得该 INode 具备该特性的功能。

Feature 的实现细节我们在 Feature 相关类小节中再做介绍, 这里先回到 InodeWithAdditionalFields 类, InodeWithAdditionalFields.features 字段就是用来保存当前 INode 拥有哪些特性的字段, 它是一个 Feature 类型的数组, 默认值是空数组 EMPTY_FEATURE。

```

private static final Feature[] EMPTY_FEATURE = new Feature[0];
protected Feature[] features = EMPTY_FEATURE;

```

InodeWithAdditionalFields 提供了向 INode 添加、删除以及查询特性的方法, 这些方法的底层还是对 features 数组的操作。以 addFeature() 方法为例, addFeature() 方法用于向当前 INode 节点添加一个新的特性, addFeature() 方法向 features 数组中添加了一个新的 Feature 元素, 代码如下:

```

protected void addFeature(Feature f) {
    int size = features.length;
    Feature[] arr = new Feature[size + 1]; // 申请一个更大的数组
    if (size != 0) {
        System.arraycopy(features, 0, arr, 0, size); // 将原 features 数组拷贝到现在数组中
    }
    arr[size] = f; // 将新的 Feature 对象添加到数组中
    features = arr; // 用新数组替换 features 引用
}

```

3. INodeDirectory 类

INodeDirectory 抽象了 HDFS 文件系统中的目录，目录是文件系统中的虚拟容器，里面保存了一组文件和其他一些目录。在 INodeDirectory 的实现中，添加了成员变量 children，用来保存目录中所有子目录项的 INode 对象。

```
// 使用一个 children 字段保存该目录中所有孩子节点的 INode 对象
private List<INode> children = null;
```

如图 3-4 所示，在 HDFS 2.6 版本的 INodeDirectory 代码中添加了 Feature 相关、Snapshot 相关的方法，我们可以将 INodeDirectory 中的方法分为如下几类。

- children 字段的增、删、改、查方法：用于向当前目录添加、删除、替换、查找子目录项等操作，包括 addChild()、removeChild()、replaceChild()、getChild()、clearChildren()、cleanSubtreeRecursively()等方法。
- 特性（Feature）相关方法：用于向当前 INodeDirectory 添加新的 Feature 对象，以及获取指定 Feature 对象的方法，包括 addDirectoryWithQuotaFeature()、getDirectoryWithQuotaFeature()、addSnapshottableFeature()、getDirectorySnapshottableFeature()、addSnapshotFeature()、getDirectoryWithSnapshotFeature()等方法。
- 快照（Snapshot）相关方法：用于向当前目录添加、删除或者更改快照等操作，包括 isSnapshottable()、getSnapshot()、setSnapshotQuota()、addSnapshot()、removeSnapshot()等方法。

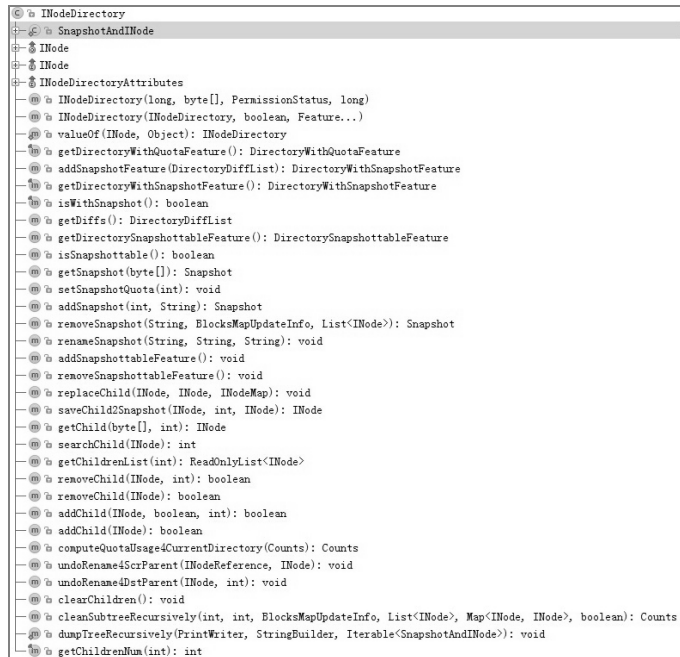


图 3-4 INodeDirectory 类结构

(1) 子目录项相关方法

`InodeDirectory` 作为目录容器，最主要的功能就是维护目录中保存的文件以及子目录，也就是维护 `children` 字段。`InodeDirectory` 提供了创建子目录项、删除子目录项以及查询子目录项等方法，这些方法的底层实现都是操作 `children` 字段，这个小节我们就学习关于 `children` 字段的操作方法。

需要特别注意的是，在 HDFS 2.6 版本的代码中引入了快照 (Snapshot) 特性。当 HDFS 管理员在当前 `InodeDirectory` 上建立了快照之后，任何对于子目录项的操作都需要在快照中进行记录。这里我们先介绍没有建立快照的子目录项操作方法，对于有快照情况下的对应方法我们在 `SnapshotFeature` 实现小节中介绍。

没有开启快照功能的 `InodeDirectory` 中子目录项的操作方法都比较简单，就是操作 `children` 这个集合。我们以 `addChild()` 和 `removeChild()` 两个方法为例，其他方法请读者自行参考源代码。`addChild()` 方法首先在 `children` 列表中找到插入位置，然后将 `Inode` 对象插入到 `children` 集合中。`removeChild()` 方法也是一样的，首先找到指定 `Inode` 对象在 `children` 集合中的位置，然后删除。`addChild()` 和 `removeChild()` 方法的代码如下：

```
public boolean addChild(Inode node) {
    // 首先找到 Inode 节点在 children 列表中的位置
    final int low = searchChildren(node.getLocalNameBytes());
    if (low >= 0) {
        return false;
    }
    // 调用 addChild() 方法将 Inode 节点插入到 children 列表的 low 位置
    addChild(node, low);
    return true;
}

public boolean removeChild(final Inode child) {
    // 找到 Inode 节点在 children 列表中的位置
    final int i = searchChildren(child.getLocalNameBytes());
    if (i < 0) {
        return false;
    }
    // 从 children 列表中删除
    final Inode removed = children.remove(i);
    Preconditions.checkNotNull(removed == child);
    return true;
}
```

(2) 特性相关方法

在 `InodeWithAdditionalFields` 类小节中，我们已经介绍过可以向 `Inode` 添加特性 (Feature)。对于 `InodeDirectory`，也就是 HDFS 目录来讲，我们可以对目录添加磁盘配额特性 (`DirectoryWithQuotaFeature`) 以及快照特性 (`DirectoryWithSnapshotFeature&DirectorySnapshottableFeature`)。添加特性方法的实现也比较简单，直接构造特性对应的 `Featrue` 对象，

然后调用 `INodeWithAdditionalFields.addFeature()` 方法将构造的 `Feature` 对象添加到 `features` 集合中即可。查找 `Feature` 对象，则直接调用 `INodeWithAdditionalFields.getFeature()` 从 `features` 集合中找到对应的 `Feature` 对象返回即可。这里以 `addDirectoryWithQuotaFeature()` 方法和 `getDirectoryWithQuotaFeature()` 方法为例，其他方法请读者自行参考源代码。

```
// 向当前目录添加磁盘配额特性
DirectoryWithQuotaFeature addDirectoryWithQuotaFeature(
    long nsQuota, long dsQuota) {
    // 构造 DirectoryWithQuotaFeature 对象，调用 addFeature() 方法添加到 features 集合中
    final DirectoryWithQuotaFeature quota = new DirectoryWithQuotaFeature(
        nsQuota, dsQuota);
    addFeature(quota);
    return quota;
}

// 获取当前目录磁盘配额特性对应的 DirectoryWithQuotaFeature 对象
public final DirectoryWithQuotaFeature getDirectoryWithQuotaFeature() {
    // 调用 getDirectoryWithQuotaFeature() 从 features 集合中查找 Feature 对象
    return getFeature(DirectoryWithQuotaFeature.class);
}
```

(3) 快照相关方法

在 HDFS 2.6 版本中，我们可以对文件系统中的任意目录建立快照，之后任何改变这个目录或者目录中子目录项的操作都会被记录在快照中，以便于日后恢复。`INodeDirectory` 中另一部分功能就是快照相关方法，主要包括 `addSnapshot()`、`getDiffs()` 等方法的实现。这一部分内容我们将在 `SnapshotFeature` 实现小节中介绍。

4. `INodeFile` 类

在文件系统目录树中，使用 `INodeFile` 类抽象一个 HDFS 文件，`INodeFile` 类继承自 `INodeWithAdditionalFields` 类。

`INodeFile` 类中保存了 HDFS 文件最重要的两个信息：文件头 `header` 字段和文件对应的数据块信息 `blocks` 字段。`header` 字段保存了当前文件有多少个副本，以及文件数据块的大小（`header` 字段的处理类似于 `INode` 中的 `permission` 字段，前 4 个比特用于保存存储策略，中间 12 个比特用于保存文件备份系数，后 48 个比特用于保存数据块大小。使用内部类 `HeaderFormat` 处理）；`blocks` 字段是一个 `BlockInfo` 类型的数组，保存了当前文件对应的所有数据块信息。`INodeFile` 定义的 `header` 字段和 `blocks` 字段的代码如下：

```
private long header = 0L; // 文件头信息
private BlockInfo[] blocks; // 文件数据块信息
```

`INodeFile` 有一个内部类 `HeaderFormat`，用于处理 `INodeFile.header` 字段。它的实现很简单，就是从 `header` 这个 `long` 类型字段中提取出前 4 个比特的存储策略信息、中间 12 个比特的文件备份系数信息，以及后 48 个比特的数据块大小信息。

blocks 字段是一个 BlockInfo 类型的数组, BlockInfo 类继承自 Block 类, 它保存了数据块与文件、数据块与数据节点的对应关系。从 BlockInfo 对象可以获得数据块所属的文件, 即文件的 INodeFile 对象, 也可以获得保存数据块副本的所有数据节点的信息, 在数据块管理小节中还会详细地介绍 BlockInfo 类的实现。

INodeFile 中的方法可以分为以下几个部分。

- 构建 (Under Construction) 特性相关方法: 当 HDFS 客户端写文件时, 该文件就处于构建状态。在早先的 HDFS 版本中, 处于构建状态中的文件是用 INodeFileUnderConstruction 类表示的; 在 HDFS 2.6 版本中, 则通过在 INodeFile 中添加 FileUnderConstructionFeature 特性来表示文件处于构建状态。这类方法包括 getFileUnderConstructionFeature()、isUnderConstruction()、toUnderConstruction()、toCompleteFile()、removeLastBlock()、setLastBlock()等。这部分代码的实现, 将在 FileUnderConstructionFeature 实现小节中进行重点介绍。
- 快照 (Snapshot) 特性相关方法: 对处于快照中的文件进行修改时, HDFS 会首先向这个文件添加 FileWithSnapshotFeature 特性, 表明这个文件在快照中。这里的方法包括 addSnapshotFeature()、getFileWithSnapshotFeature()、recordModification()、getDiffs()等。这部分代码的实现, 将在 SnapshotFeature 实现小节中介绍。
- 其他方法: 主要包括获取和修改 header 字段信息的方法, 以及对 blocks 数组字段进行操作的方法。这部分代码的实现比较简单, 我们就不再赘述了。

5. INodeReference

当 HDFS 文件/目录处于某个快照中, 并且这个文件/目录被重命名或者移动到其他路径时, 该文件/目录就会存在多条访问路径。INodeReference 及其子类就是为了解决这个问题而产生的。

这里我们举一个例子, 如图 3-5 所示, /abc 是 HDFS 文件系统中的一个普通目录, 管理员为/abc 目录建立了一个快照 s0, /abc 目录下有一个文件 foo。根据快照功能的定义, 用户可以通过路径/abc/foo 以及/abc/snapshot/s0/foo 访问 foo 文件。

当用户将/abc/foo 文件重命名为/xyz/bar 时, 通过快照路径/abc/snapshot/s0/foo 将无法访问 foo 文件, 如图 3-6 所示, 这种情况是不符合快照规范的。

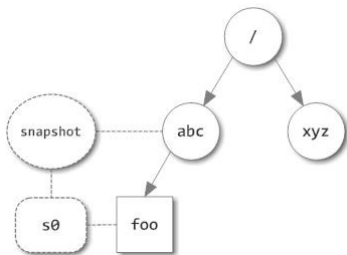


图 3-5 带有快照的文件

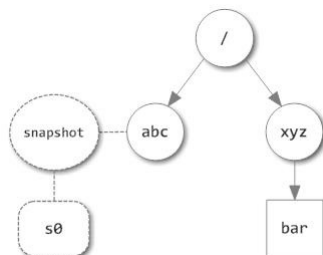


图 3-6 重命名后的文件

为了解决这个问题，HDFS 引入了 `InodeReference` 类。图 3-7 给出了 `InodeReference` 及其子类的继承关系图，这里的 `WithName`、`WithCount`、`DstReference` 都是 `InodeReference` 的子类，同时也是 `InodeReference` 的内部类。`WithName` 对象用于替代重命名操作前源路径中的 `Inode` 对象，`DstReference` 对象则用于替代重命名操作后目标路径中的 `Inode` 对象。`WithName` 和 `DstReference` 共同指向了一个 `WithCount` 对象，`WithCount` 对象则指向了文件系统目录树中真正的 `Inode` 对象。

图 3-8 给出了使用 `InodeReference` 后的文件目录树，当进行重命名操作时，`Namenode` 会在 `/abc` 目录下添加 `InodeReference.WithName` 节点替代重命名前的 `foo` 节点，在 `/xyz` 目录下添加 `InodeReference.DstReference` 节点替代重命名后的 `bar` 节点。`InodeReference.WithName` 以及 `InodeReference.DstReference` 则共同指向一个 `InodeReference.withCount` 节点，`InodeReference.withCount` 节点指向真实的 `Inode` 节点 `bar`。这样，无论用户是通过 `/xyz/bar` 路径还是 `/abc/snapshot/s0/foo` 快照路径访问文件，都可以通过获取到 `withCountReference` 对象的引用找到真正的 `Inode` 节点 `bar`，也就解决了这个问题。

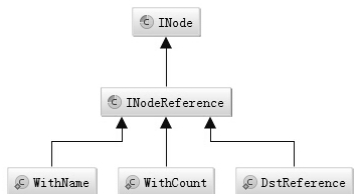


图 3-7 InodeReference 继承关系图

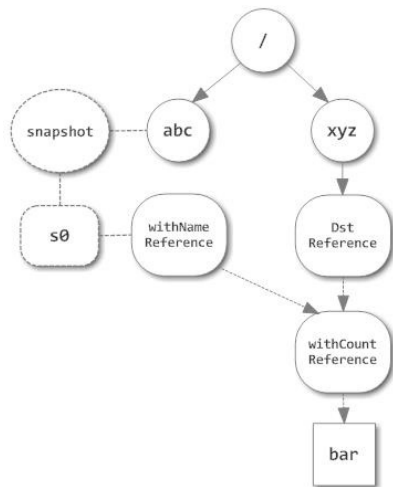


图 3-8 使用 InodeReference 后的文件目录树

了解了 `InodeReference` 的概念后，我们再来看 `InodeReference` 的代码实现。`InodeReference` 是一个抽象类，它扩展自 `Inode` 类，所以 `InodeReference` 及其子类是可以添加到文件系统目录树中以替代原有的 `InodeFile` 节点的。`InodeReference` 定义了 `referred` 字段，这个字段非常重要，用于保存当前 `InodeReference` 类指向的 `Inode` 节点。例如，对于 `InodeReference` 的子类 `WithName` 和 `DstReference` 来说，`referred` 字段就指向了 `WithCount` 对象；而对于 `WithCount` 来说，`referred` 字段则指向了文件系统目录树中真正的 `Inode` 对象。`InodeReference` 抽象类还定义了 `getReferredInode()` 方法，在文件系统目录树的操作中，如果判断当前节点是一个引用节点，则会调用 `getReferredInode()` 方法获取 `InodeReference` 类指向的 `Inode` 对象。

```
public abstract class InodeReference extends Inode {
    private Inode referred; // 指向的 Inode 节点
}
```



```

public INodeReference(INode parent, INode referred) {
    super(parent);
    this.referred = referred;
}
public final INode getReferredINode() { // 获取指向的 INode 节点
    return referred;
}
public final void setReferredINode(INode referred) {
    this.referred = referred;
}
//...
}

```

了解了 `INodeReference` 抽象类的代码实现后,我们再来看 `WithCount` 类的实现。`WithCount` 类定义了一个集合字段 `withNameList` 用于保存所有指向这个 `WithCount` 对象的 `WithName` 对象的集合。`WithCount` 类还定义了 `addReference()` 方法,任何指向 `WithCount` 对象的 `WithName` 对象以及 `DstReference` 对象都需要调用这个方法添加指向关系。对于指向这个 `WithCount` 对象的 `DstReference` 对象, `addReference()` 方法会将这个对象设置为自己的父 `INode` 节点 (通过 `INode.parent` 字段);而对于 `WithName` 对象, `addReference()` 方法则将这个对象放入 `withNameList` 集合中保存。

```

public static class WithCount extends INodeReference {
    // 保存所有指向这个 WithCount 对象的 WithName 对象的集合
    private final List<WithName> withNameList = new ArrayList<WithName>();

    public WithCount(INodeReference parent, INode referred) {
        super(parent, referred); // 调用父类的构造方法,指向文件系统目录树中的 INode
        Preconditions.checkArgument(!referred.isReference());
        referred.setParentReference(this); // 设置真实 INode 的父节点为当前 WithCount 对象
    }

    public void addReference(INodeReference ref) {
        if (ref instanceof WithName) { // 如果是 WithName 对象,则加入 withNameList
            WithName refWithName = (WithName) ref;
            int i = Collections.binarySearch(withNameList, refWithName,
                WITHNAME_COMPARATOR);
            Preconditions.checkState(i < 0);
            withNameList.add(-i - 1, refWithName);
        } else if (ref instanceof DstReference) { // 如果是 DstReference 对象,则设置为父节点
            setParentReference(ref);
        }
    }
    // ...
}

```

了解了 `WithCount` 对象的实现后,我们再来看 `WithName` 和 `DstReference`,这两个类的实现都比较简单。`WithName` 类定义了 `name` 字段用于保存重命名前文件的名称,同时定义了 `lastSnapshotId` 字段用于保存 `WithName` 对象构造时源路径的快照版本号。`DstReference` 类的实

现就更简单了，只定义了一个 `dstSnapshotId` 字段用于保存重命名操作前目标路径的最新快照的版本号。`WithName` 和 `DstReference` 在构造时都会调用父类的构造方法指向 `WithCount` 对象，同时还会调用 `WithCount.addReference()` 方法配置 `WithCount` 对象。

```
public static class WithName extends INodeReference {
    private final byte[] name;    // 重命名前的文件名
    private final int lastSnapshotId;
    public WithName(INodeDirectory parent, WithCount referred, byte[] name,
        int lastSnapshotId) {
        super(parent, referred); // 调用父类构造方法，指向 WithCount 节点
        this.name = name;
        this.lastSnapshotId = lastSnapshotId;
        referred.addReference(this); // 调用 WithCount.addReference()
    }
    // ...
}

public static class DstReference extends INodeReference {
    private final int dstSnapshotId;
    public DstReference(INodeDirectory parent, WithCount referred,
        final int dstSnapshotId) {
        super(parent, referred); // 调用父类构造方法，指向 WithCount 节点
        this.dstSnapshotId = dstSnapshotId;
        referred.addReference(this); // 调用 WithCount.addReference()
    }
    // ...
}
```

了解了 `INodeReference` 的代码实现后，我们再来看 `Namenode` 重命名操作的代码实现。建立 `INodeReference` 节点操作的入口是 `FsDirectory.RenameOperation`，只有在进行 `rename` 操作时，也就是将一个快照中的 `INode` 节点重命名为另一个路径下的 `INode` 节点时才有可能创建 `INodeReference` 节点。这里我们看一下 `RenameOperation` 的具体操作。首先看一下 `RenameOperation` 的构造方法，构造方法首先判断 `Rename` 操作的源节点是否在快照中，如果在快照中，则调用 `INodeDirectory.replaceChild4ReferenceWithName()` 方法构造 `INodeReference` 对象，并将 `INodeDirectory` 中的 `srcINode` 对象全部替换为该对象（需要特别注意的是，如果 `srcINode` 存在于快照 `diff` 对象的 `c-list` 列表中，也是需要替换的）。

```
private RenameOperation(String src, String dst, INodesInPath srcIIP, INodesInPath
dstIIP)
    throws QuotaExceededException {
    this.srcIIP = srcIIP;
    this.dstIIP = dstIIP;
    this.src = src;
    this.dst = dst;
    srcChild = srcIIP.getLastINode();
    srcChildName = srcChild.getLocalNameBytes();
    isSrcInSnapshot = srcChild.isInLatestSnapshot(
```

```

        srcIIP.getLatestSnapshotId());
srcChildIsReference = srcChild.isReference();
srcParent = srcIIP.getInode(-2).asDirectory();

if (isSrcInSnapshot) {
    srcChild.recordModification(srcIIP.getLatestSnapshotId());
}

srcRefDstSnapshot = srcChildIsReference ? srcChild.asReference()
    .getDstSnapshotId() : Snapshot.CURRENT_STATE_ID;
oldSrcCounts = Quota.Counts.newInstance();
// 如果源节点在快照中
if (isSrcInSnapshot) {
    // 调用 replaceChild4ReferenceWithName() 方法构造 WithName 和 WithCount 对象
    final INodeReference.WithName withName = srcIIP.getInode(-2).asDirectory()
        .replaceChild4ReferenceWithName(srcChild, srcIIP.getLatestSnapshotId());
    withCount = (INodeReference.WithCount) withName.getReferredINode();
    srcChild = withName;
    srcIIP.setLastINode(srcChild);
    withCount.getReferredINode().computeQuotaUsage(oldSrcCounts, true);
} else if (srcChildIsReference) {
    withCount = (WithCount) srcChild.asReference().getReferredINode();
} else {
    withCount = null; // 否则将 withCount 设置为 null, 也就是普通的重命名操作
}
}

```

完成了对 `withCount` 的构造之后, 就可以调用 `addSourceToDestination()` 方法将源节点或者 `dst` 节点添加到目标路径了。这里分为两种情况: `withCount==null`, 也就是普通的重命名操作, 则不需要使用 `INodeReference` 机制, 在这种情况下直接将源 `INode` 节点添加到目标路径即可; `withCount!=null`, 也就是需要使用 `INodeReference` 机制的情况, 这里则构造 `DstReference` 对象, 然后将这个 `DstReference` 对象添加到目标路径即可。这里注意, `DstReference` 对象的构造方法会将 `DstReference.referred` 字段设置为 `WithCount` 对象, 然后设置 `WithCount` 对象的父节点为当前 `DstReference` 对象。

```

boolean addSourceToDestination() {
    final INode dstParent = dstIIP.getInode(-2);
    srcChild = srcIIP.getLastINode();
    final byte[] dstChildName = dstIIP.getLastLocalName();
    final INode toDst;
    if (withCount == null) {
        srcChild.setLocalName(dstChildName); // 普通情况, 则直接添加源节点
        toDst = srcChild;
    } else { // 要使用 INodeReference 机制的情况, 则构造 DstReference 对象
        withCount.getReferredINode().setLocalName(dstChildName);
        int dstSnapshotId = dstIIP.getLatestSnapshotId();
        toDst = new INodeReference.DstReference(
            dstParent.asDirectory(), withCount, dstSnapshotId);
    }
}

```

```
}  
    return addLastINodeNoQuotaCheck(dstIIP, toDst); // 将 toDst 节点添加到目标路径  
}
```

3.1.2 Feature 相关类

在 HDFS 2.6 版本中引入了特性 (Feature) 这个概念, HDFS 定义了 `INode.Feature` 接口抽象 `INode` 特性的根接口, 如图 3-9 所示, `INode` 的所有特性都实现了这个接口, 每个特性都对应一个 `Feature` 子类, 包括:

- `DirectoryWithSnapshotFeature`: 带有快照的目录特性。
- `DirectorySnapshottableFeature`: 可以添加快照的目录特性。
- `FileWithSnapshotFeature`: 带有快照的文件特性。
- `DirectoryWithQuotaFeature`: 支持磁盘配额的目录特性。
- `FileUnderConstructionFeature`: 正在构建的文件特性。
- `XAttrFeature`: 支持文件系统扩展属性的特性。
- `AclFeature`: 安全特性, 我们暂不介绍。

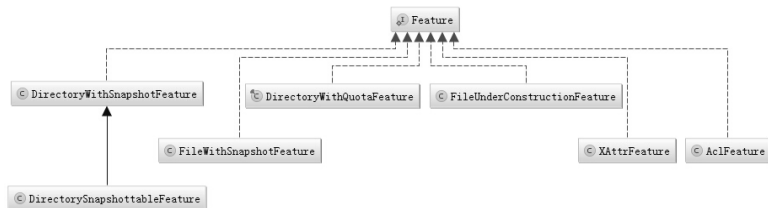


图 3-9 Feature 接口继承关系图

本节就介绍快照特性对应的 `SnapshotFeature` 类, 以及构建中特性对应的 `FileUnderConstructionFeature` 类。

1. SnapshotFeature 实现

HDFS 2.6 版本发布了快照功能, HDFS 管理员可以对 HDFS 中的任意目录建立快照 (Snapshot)。快照是一个文件系统, 或者是文件系统中某个目录在某一时刻的镜像。在目标目录的快照创建之后, 不论目标目录发生任何变化, 或者目标目录中保存的子目录项发生任何变化, 都可以通过快照找回快照建立时目标目录的所有文件以及目录结构。本节我们就学习 HDFS 快照功能的实现。

在为目标目录创建快照之前, 需要先执行 “`hdfs dfsadmin -allowSnapshot`” 命令启用目标目录的快照功能。在目标目录启用快照功能后, 该目录并不会自动执行快照保存操作, 管理员还需要执行 “`hdfs dfs -createSnapshot`” 命令创建快照。需要注意的是, 同一个目录可以创建多个快照, 不同的快照通过名字来区分。

我们首先以一个例子来分析整个快照功能执行的流程。如图 3-10 所示, 左边的树是一棵

普通的文件系统目录树。当管理员执行“hdfs dfsadmin-allowSnapshot”命令在目录 a 上开启快照功能后，HDFS 会创建一个 DirectorySnapshottableFeature 对象，然后将这个新创建的 Feature 对象添加到目录 a 对应的 INodeDirectory 对象的 features 集合中，如右图所示。

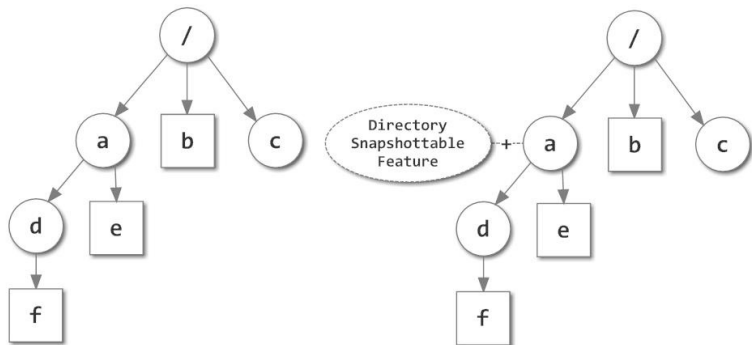


图 3-10 开启快照功能

成功开启目录 a 的快照功能后，管理员就可以执行“hdfs dfs-createSnapshot”命令在目录 a 下创建一个快照 s1。如图 3-11 所示，HDFS 会创建一个 DirectoryDiff 对象记录 s1 快照创建之后目录 a 上执行的所有操作，并将这个新创建的 DirectoryDiff 对象添加到 DirectorySnapshottableFeature 的 DirectoryDiffList 对象中保存。DirectoryDiff 会通过持有一个 ChildrenDiff 对象记录目录 a 的子目录项的变化情况，ChildrenDiff 对象的 c-list 集合保存了快照创建之后目录 a 下所有新添加的文件或者目录，ChildrenDiff 对象的 d-list 集合则保存了快照创建之后从目录 a 删除的文件或者目录。

如图 3-12 所示，成功创建快照 s1 后，当我们从目录 a 删除文件 e 时，文件 e 并不会直接从 INodeDirectory 中完全删除，而是暂时保存在快照 s1 对应的 DirectoryDiff 对象的 d-list 集合中。

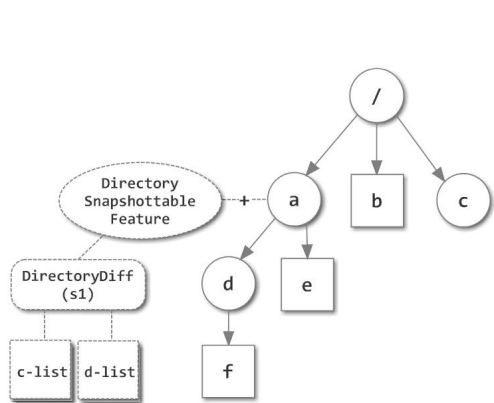


图 3-11 建立快照 s1

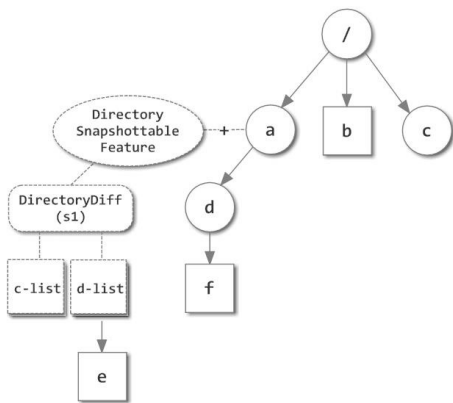


图 3-12 删除文件 e

删除文件 e 之后，我们在目录 a 下再创建一个快照 s2，然后向目录 a 添加一个文件 g。如

图 3-13 所示，HDFS 会在 `DirectorySnapshottableFeature` 对象上添加一个新的 `DirectoryDiff` 对象记录快照 s2 创建之后在目录 a 上进行的所有操作，这样快照 s1 对应的 `DirectoryDiff` 对象就记录了快照 s1 和快照 s2 之间目录 a 上执行的操作。成功创建 `DirectoryDiff` 对象之后，HDFS 将新添加的文件 g 放入 `DirectoryDiff` 的 c-list 集合中保存。当用户在快照 s1 上检索文件 e 时，由于文件 e 保存在快照的 d-list 中，所以文件可以正常返回。当用户在快照 s2 上检索文件 g 时，由于文件 g 在 c-list 中，也就是新创建的文件，HDFS 会返回空，表明快照 s2 创建时 a 目录下并没有这个文件。

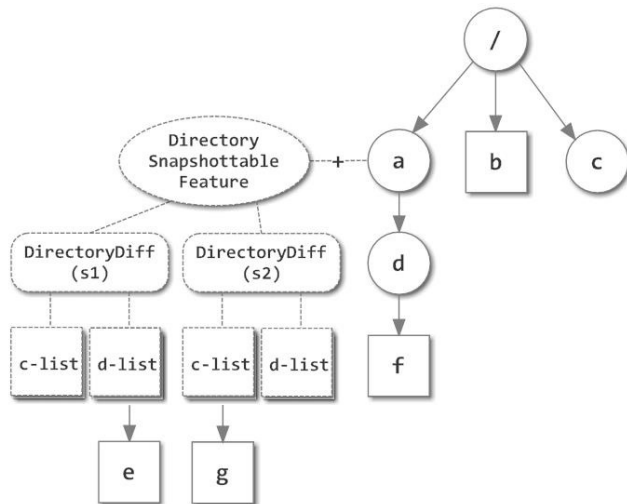


图 3-13 建立快照 s2

至此，一个完整的快照流程就介绍完了。在下面的两个小节中我们介绍实现快照功能的类，以及快照流程的代码实现。

(1) 快照相关类

我们在 `Inode` 相关类小节中介绍了，为 `Inode` 添加快照特性需要创建一个 `Feature` 类型的对象，然后将这个 `Feature` 对象添加到 `Inode` 的 `features` 字段中保存。HDFS 中定义了 `DirectoryWithSnapshotFeature` 和 `DirectorySnapshottableFeature` 两个类来描述目录的快照特性。

`DirectorySnapshottableFeature` 描述了目录开启快照功能的特性，`DirectoryWithSnapshotFeature` 则描述了目录拥有快照的特性。HDFS 的单个目录可以定义多个快照，快照和快照之间进行的所有操作使用 `DirectoryDiff` 类来记录。`DirectoryDiff` 类记录了 HDFS 目录的某个快照版本和上一个快照版本之间进行的所有操作，`DirectoryDiff` 类使用 `ChildrenDiff` 类记录该 HDFS 目录的子目录项集合 `children` 的变化情况，`ChildrenDiff` 的 c-list 保存了快照建立之后新创建的子目录项，d-list 则保存了删除的子目录项。

由于一个 HDFS 目录可以拥有多个快照，单个快照和上一个快照之间进行的操作是通过一个 `DirectoryDiff` 对象来记录的，所以 HDFS 目录需要持有多个 `DirectoryDiff` 对象来记录所

有快照之间进行的操作。HDFS 定义了 `DirectoryDiffList` 类来保存这些 `DirectoryDiff` 对象，`DirectoryDiffList` 底层持有一个 `DirectoryDiff` 的集合，集合中的所有 `DirectoryDiff` 元素是按照快照 ID 进行排序的，并且每个 `DirectoryDiff` 对象都持有下一个 `DirectoryDiff` 对象的引用，类似于链表的功能。

快照功能涉及的所有类如图 3-14 所示，DirectoryWithSnapshotFeature 类通过持有一个 DirectoryDiffList 对象记录了 HDFS 目录中所有快照之间进行的全部操作；而 DirectoryDiffList 底层则保存了多个 DirectoryDiff 对象，每个 DirectoryDiff 对象记录了目录的一个快照版本和上一个快照版本之间进行的操作；DirectoryDiff 底层则通过持有一个 ChildrenDiff 对象来记录目录中所有子目录项的变化操作。下面我们将一一介绍这些类的具体实现。

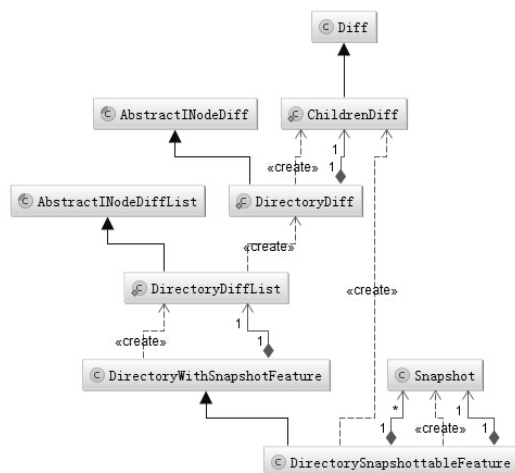


图 3-14 快照功能涉及的类

Diff 类

我们首先看一下类图中的第一个类 **Diff**。**Diff** 类用来描述一个集合的当前状态和上一个状态之间的不同。**Diff** 类中有两个列表：**c-list** 记录了集合在两个状态之间新创建的元素；**d-list** 记录了集合在两个状态之间删除的元素。使用这两个列表，**Diff** 类可以记录集合上一个状态与当前状态之间所有新创建、删除与修改的元素。

Diff 类的算法很简单：对于新创建的元素，放入 **c-list** 中；对于新删除的元素，放入 **d-list** 中；对于修改的元素，则同时放入 **c-list** 和 **d-list** 中。

ChildrenDiff 类

ChildrenDiff 类用于描述 INodeDirectory 的子目录项集合 children 字段的当前状态与上一个快照版本状态之间的差异。

ChildrenDiff 扩展自 Diff 类，从类定义代码我们可以看出，ChildrenDiff 描述的是 INode 集合状态间的差异，而 INodeDirectory 的 children 字段正好是一个 List<INode>类型，也就是

目录中所有子目录项（包括文件和子目录）的 `INode` 集合。由此我们可以知道，`ChildrenDiff` 的 `c-list`（创建列表）和 `d-list`（删除列表）存放的就是 `INodeDirectory` 在当前状态和上一个快照状态之间新创建的子目录项以及删除的子目录项。`ChildrenDiff` 类的定义如下：

```
static class ChildrenDiff extends Diff<byte[], INode> {
    ChildrenDiff() {}

    private ChildrenDiff(final List<INode> created, final List<INode> deleted) {
        super(created, deleted);
    }
    ...
}
```

AbstractINodeDiff 类

`AbstractINodeDiff` 类抽象了 `INode` 在两个快照版本之间的状态差异。如下代码所示，`AbstractINodeDiff` 类中包含一个 `posteriorDiff` 字段，用于链接下一个快照版本与当前快照版本之间差异的 `AbstractINodeDiff` 对象，如果 `posteriorDiff` 字段为空，则表明 `INode` 的当前状态为最新状态。

```
abstract class AbstractINodeDiff<N extends INode,
                                A extends INodeAttributes,
                                D extends AbstractINodeDiff<N, A, D>>
    implements Comparable<Integer> {

    /** 当前 AbstractINodeDiff 对象对应的快照 ID */
    private int snapshotId;
    /** 快照所对应的 INode 的数据，如果没有变化则为 null */
    A snapshotINode;
    /** 下一个快照版本与当前快照版本之间差异的 AbstractINodeDiff */
    private D posteriorDiff;

    // .....
}
```

DirectoryDiff 类

`DirectoryDiff` 是 `DirectoryWithSnapshotFeature` 的内部类，它扩展自 `AbstractINodeDiff` 类。`AbstractINodeDiff` 类用于抽象 `INode` 在两个快照版本之间的状态差异，由此可以知道 `DirectoryDiff` 就是用于描述 `INodeDirectory` 在两个快照版本之间的状态差异的。

如下代码所示，`DirectoryDiff` 类通过持有一个 `ChildrenDiff` 对象来记录 `INodeDirectory` 的当前快照与上一个快照间的子目录项集合的变化情况，也就是 `children` 字段的变化情况。

```
public static class DirectoryDiff extends
    AbstractINodeDiff<INodeDirectory, INodeDirectoryAttributes, DirectoryDiff> {
    /** 快照创建时，INodeDirectory 子目录项的数目 */
    private final int childrenSize;
    /** 子目录项 diff */
}
```



```

private final ChildrenDiff diff;
private boolean isSnapshotRoot = false;

private DirectoryDiff(int snapshotId, INodeDirectory dir) {
    super(snapshotId, null, null);
    // 获取快照建立时，子目录项的个数
    this.childrenSize = dir.getChildrenList(Snapshot.CURRENT_STATE_ID).size();
    // 构造一个新的 ChildrenDiff 对象，记录 INodeDirectory.children 字段的变化
    this.diff = new ChildrenDiff();
}
// ...
}

```

AbstractINodeDiffList 类

AbstractINodeDiffList 维护了一个 AbstractINodeDiff 的集合列表，用于保存一个 INode 拥有的所有 AbstractINodeDiff 对象，也就是记录了 INode 所有快照间差异的列表。

如下代码所示，AbstractINodeDiffList 定义了一个 diffs 集合字段，diffs 保存了 INode 所有快照间的 AbstractINodeDiff 对象，所有 AbstractINodeDiff 对象是按照快照 ID 排序的。AbstractINodeDiffList 的方法主要用于维护 diffs 这个数据结构。

```

abstract class AbstractINodeDiffList<N extends INode,
                                     A extends INodeAttributes,
                                     D extends AbstractINodeDiff<N, A, D>>
    implements Iterable<D> {
    /** 快照的 Diff 列表，这个列表按照快照 ID 排序 */
    private final List<D> diffs = new ArrayList<D>();
}

```

DirectoryDiffList 类

DirectoryDiffList 扩展自 AbstractINodeDiffList，维护了 INodeDirectory 所有快照间差异的集合，也就是 DirectoryDiff 对象的集合。DirectoryDiffList 对象的定义代码如下：

```

public static class DirectoryDiffList
    extends AbstractINodeDiffList<INodeDirectory, INodeDirectoryAttributes, DirectoryDiff>
{
}

```

DirectoryWithSnapshotFeature 类

DirectoryWithSnapshotFeature 类用于存储和处理指定目录下建立的所有快照，也就是记录目录快照之间进行的添加、删除、修改目录项的操作。需要注意的是，DirectoryDiff、DirectoryDiffList 以及 ChildrenDiff 类都是 DirectoryWithSnapshotFeature 类的内部类。

DirectoryWithSnapshotFeature 类持有一个 DirectoryDiffList 类型的字段 diffs，用于保存当前目录以及它的子目录项在不同快照间的所有差异。DirectoryWithSnapshotFeature 类的构造方法也非常简单，就是判断传入的 DirectoryDiffList 是否为 null，不为 null 则构造一个新的 DirectoryDiffList。

```

private final DirectoryDiffList diffs;

```

```
public DirectoryWithSnapshotFeature(DirectoryDiffList diffs) {  
    this.diffs = diffs != null ? diffs : new DirectoryDiffList();  
}
```

在老版本的 HDFS 代码中，向 `INodeDirectory` 中添加、删除以及修改子目录项的操作只需要更改 `INodeDirectory.children` 字段即可。添加了快照机制之后，对于子目录项的修改都需要由 `DirectoryWithSnapshotFeature` 代理。考虑下面这种情况，HDFS 管理员为目录 `/tmp/dir` 建立快照 `s0`，则 `/tmp/dir` 对应的 `INodeDirectory` 会首先被添加 `DirectoryWithSnapshotFeature` 特性，并在 `DirectoryWithSnapshotFeature.directoryDiffList` 字段中添加一个当前快照 `s0` 对应的 `DirectoryDiff` 对象。这个对象用于保存 `s0` 建立之后对 `/tmp/dir` 目录的添加、删除、修改子目录项等操作。所以 `INodeDirectory` 中所有对于 `children` 字段的修改方法都会由 `DirectoryWithSnapshotFeature` 中对应的方法代理，比如 `addChild()`、`removeChild()` 方法等。

通过前面章节的介绍我们知道，`DirectoryDiff` 对象会将当前目录中新创建的子目录项放入 `DirectoryDiff.childrenDiff` 字段的 `c-list` 中，而将删除的子目录项放入 `DirectoryDiff.childrenDiff` 字段的 `d-list` 中。还是继续刚才的例子，当管理员成功为 `/tmp/dir` 建立了一个名为 `s0` 的快照后，用户向 `/tmp/dir` 目录中新写入一个 `syslog1` 文件时，这个文件会被添加到 `s0` 快照对应的 `DirectoryDiff` 对象的 `c-list` 中。当用户通过 `s0` 快照路径查找 `syslog1` 文件时，由于 `syslog1` 文件在 `DirectoryDiff` 的 `c-list` 中，也就是该文件是快照建立之后新创建的文件，HDFS 会返回未找到该文件。当用户正常查找 `syslog1` 文件而不通过快照路径查找时，HDFS 将会直接返回 `syslog1` 文件。删除操作与添加操作同理，这里就不再详细解释了。

了解清楚了流程之后，再看 `DirectoryWithSnapshotFeature` 的 `addChild()` 以及 `removeChild()` 方法就比较简单了。这里以 `addChild()` 方法为例，`addChild()` 方法首先从 `DirectoryDiffList` 中获取当前快照对应的 `DirectoryDiff` 对象，然后获取这个 `DirectoryDiff` 对象中保存的 `ChildrenDiff` 对象。之后调用 `ChildrenDiff.create()` 方法将这个新添加的 `INode` 对象放入 `ChildrenDiff` 的 `c-list` 中。成功在 `DirectoryDiff` 对象中记录修改操作后，`DirectoryWithSnapshotFeature` 的代理工作就完成了，这时候 `addChild()` 方法会调用 `INodeDirectory.add()` 方法，将新创建的 `INode` 对象放入 `INodeDirectory.children` 字段中。

```
public boolean addChild(INodeDirectory parent, INode inode,  
    boolean setModTime, int latestSnapshotId) throws QuotaExceededException {  
    // 根据 latestSnapshotId 获取 DirectoryDiff 的 ChildrenDiff 字段  
    ChildrenDiff diff = diffs.checkAndAddLatestSnapshotDiff(latestSnapshotId,  
        parent).diff;  
    // 在 ChildrenDiff 的 c-list 中添加新创建的 INode 对象  
    int undoInfo = diff.create(inode);  
  
    // 调用 INodeDirectory.add() 方法将新创建的 INode 对象加入 INodeDirectory.children 字段  
    final boolean added = parent.addChild(inode, setModTime,  
        Snapshot.CURRENT_STATE_ID);  
    if (!added) {  
        diff.undoCreate(inode, undoInfo);  
    }  
    return added;  
}
```

除了添加和删除子目录项外，获取一个目录在指定快照状态下的所有子目录项也是一个很重要的操作。还是考虑刚才的例子，HDFS 目录/tmp/dir 下有三个文件 a、b、c，这时建立快照 s0，然后删除文件 a、b，再建立快照 s1。当我们通过 s1 快照获取 dir 目录的子目录项时，应当返回文件 c。通过 s0 快照获取时，应该返回文件 a、b、c。这个逻辑是由 `getChildrenList()` 方法实现的，`getChildrenList()` 方法首先从 `DirectoryDiffList` 中获取指定快照 ID 对应的 `DirectoryDiff` 对象，然后在该 `DirectoryDiff` 对象上调用 `getChildrenList()` 方法。

```
public ReadOnlyList<INode> getChildrenList(INodeDirectory currentINode,
    final int snapshotId) {
    // 获取快照对应的 DirectoryDiff 对象
    final DirectoryDiff diff = diffs.getDiffById(snapshotId);
    // 调用 DirectoryDiff.getChildrenList() 方法获取该快照下的所有子目录项
    return diff != null ? diff.getChildrenList(currentINode) : currentINode
        .getChildrenList(Snapshot.CURRENT_STATE_ID);
}
```

`DirectoryDiff.getChildrenList()` 方法会将目录的指定快照状态与当前状态间的所有 `ChildrenDiff` 合并成一个 `ChildrenDiff` 对象 `combined`，然后调用 `combined.apply2Current()` 方法通过目录当前状态以及 `combined` 逆推出快照建立时目录的状态，最后返回。

```
private ReadOnlyList<INode> getChildrenList(final INodeDirectory currentDir) {
    return new ReadOnlyList<INode>() {
        // ...
        private List<INode> initChildren() {
            if (children == null) {
                // 将指定快照状态与当前目录状态间的所有操作合并成一个 ChildrenDiff 对象
                final ChildrenDiff combined = new ChildrenDiff();
                for (DirectoryDiff d = DirectoryDiff.this; d != null;
                    d = d.getPosterior()) {
                    combined.combinePosterior(d.diff, null);
                }
                // 在 combined 对象上调用 apply2Current() 方法逆推出快照建立时的目录状态
                children = combined.apply2Current(ReadOnlyList.Util.asList(
                    currentDir.getChildrenList(Snapshot.CURRENT_STATE_ID)));
            }
            // 返回快照建立时目录中的所有子目录项
            return children;
        }
        // ...
    };
}
```

了解了 `DirectoryWithSnapshotFeature.getChildrenList()` 方法的实现后，我们再学习 `DirectoryWithSnapshotFeature.getChild()` 方法，这个方法用于获取当前目录在指定快照状态下的一个 `INode` 对象。`getChild()` 方法首先获取 `snapshotId` 对应的 `DirectoryDiff` 对象，然后在 `DirectoryDiff` 对象上调用 `getChild()` 方法获取这个快照下的指定 `INode` 对象。

```
public INode getChild(INodeDirectory currentINode, byte[] name,
    int snapshotId) {
    final DirectoryDiff diff = diffs.getDiffById(snapshotId);
    return diff != null ? diff.getChild(name, true, currentINode)
        : currentINode.getChild(name, Snapshot.CURRENT_STATE_ID);
}
```

DirectoryDiff.getChild()方法会从目录的快照版本向后遍历，依次获得快照间保存差异操作的 DirectoryDiff 对象（DirectoryDiff 的 posteriorDiff 字段保存了下一个快照版本的 DirectoryDiff 对象的引用），然后在 DirectoryDiff 对象的 diff 字段（ChildreDiff）上调用 accessPrevious()方法判断差异操作中是否保存了要查找的 INode 对象，如果保存了则返回这个 INode 对象。如果在快照版本与当前版本之间的所有 DirectoryDiff 对象中都没有这个 INode 的信息，则调用 INodeDirectory.getChild()方法从当前状态的目录中获取这个 INode 对象并返回。

```
INode getChild(byte[] name, boolean checkPosterior,
    INodeDirectory currentDir) {
    for(DirectoryDiff d = this; ; d = d.getPosterior()) {
        // 调用 ChildrenDiff.accessPrevious()方法获取 INode 信息
        final Container<INode> returned = d.diff.accessPrevious(name);
        if (returned != null) {
            // 如果在两个版本间的 DirectoryDiff 中记录了这个 INode 对象，则返回
            return returned.getElement();
        } else if (!checkPosterior) {
            return null;
        } else if (d.getPosterior() == null) {
            // 所有快照版本间的 DirectoryDiff 对象中都没有记录，则在当前目录上查找
            return currentDir.getChild(name, Snapshot.CURRENT_STATE_ID);
        }
    }
}
```

DirectorySnapshottableFeature 类

DirectorySnapshottableFeature 类用于描述开启快照功能的目录，这个类是 DirectoryWithSnapshotFeature 的子类，相对于父类它添加了一些描述在这个目录下创建的所有快照的字段。

如图 3-15 所示，DirectorySnapshottableFeature 中最重要的字段就是 snapshotsByNames，它是一个 Snapshot 类的集合，用于保存当前目录下创建的所有快照。Snapshot 类用于描述一个新创建的快照，包括了快照 ID、快照根节点、快照名等信息。

DirectorySnapshottableFeature 定义的方法也多是处理 snapshotsByNames 这个字段的，方法的作用通过方法名就可以看出来，实现也比较简单。这里我们重点介绍一下 computeDiff()方法，其他方法比较简单，就不再详细介绍了。

DirectorySnapshottableFeature	
SNAPSHOT_LIMIT	int
snapshotsByNames	List<Snapshot>
snapshotQuota	int
getNumSnapshots()	int
searchSnapshot(byte[])	int
getSnapshot(byte[])	Snapshot
getSnapshotById(int)	Snapshot
getSnapshotList()	ReadOnlyList<Snapshot>
renameSnapshot(String, String, String)	void
getSnapshotQuota()	int
setSnapshotQuota(int)	void
addSnapshot(Snapshot)	void
addSnapshot(INodeDirectory, int, String)	Snapshot
removeSnapshot(INodeDirectory, String, BlocksMapUpdateInfo, List<INode>)	void
computeContentSummary(INodeDirectory, ContentSummaryComputationContext)	Context
computeDiff(INodeDirectory, String, String)	SnapshotDiffInfo
getSnapshotByName(INodeDirectory, String)	Snapshot
computeDiffRecursively(INodeDirectory, INode, List<byte[]>, SnapshotDiffInfo)	void
findRenameTargetPath(INodeDirectory, WithName, int)	byte[][]
toString()	String
dumpTreeRecursively(INodeDirectory, PrintWriter, StringBuilder, int)	void

图 3-15 DirectorySnapshottableFeature 类结构

`computeDiff()` 方法用于计算指定目录的两个快照版本之间的差异，它会返回一个 `SnapshotDiffInfo` 类记录哪些子目录项发生了变化、哪些子目录项被重命名、发生变化子目录项的 `ChildrenDiff` 对象的引用等信息。`computeDiff()` 方法首先在 `snapshotsByNames` 集合中查找两个快照的 `Snapshot` 对象的引用，然后调用 `computeDiffRecursively()` 方法递归计算两个快照之间的差异。

```
SnapshotDiffInfo computeDiff(final INodeDirectory snapshotRoot,
    final String from, final String to) throws SnapshotException {
    // 从 snapshotsByNames 集合中获取两个快照的引用
    Snapshot fromSnapshot = getSnapshotByName(snapshotRoot, from);
    Snapshot toSnapshot = getSnapshotByName(snapshotRoot, to);

    if (from.equals(to)) {
        return null;
    }
    // 构造 SnapshotDiffInfo 对象，用于保存两个快照之间的差异
    SnapshotDiffInfo diffs = new SnapshotDiffInfo(snapshotRoot, fromSnapshot,
        toSnapshot);
    // 调用 computeDiffRecursively(), 将结果放入 SnapshotDiffInfo 对象中
    computeDiffRecursively(snapshotRoot, snapshotRoot, new ArrayList<byte[]>(),
        diffs);
    // 返回 diffs
    return diffs;
}
```

`computeDiffRecursively()` 方法首先判断当前目录或者文件是否发生了改变，如果发生了改变，则记录到 `SnapshotDiffInfo` 中。判断完当前目录后，递归调用判断当前目录的子目录项的改变情况，这里需要注意，对于在新版本的快照中已经删除的目录，不需要递归判断该目录的子目录项。那么对于目录来说，如何计算两个快照版本之间发生的变化呢？这里调用了

`DirectoryWithSnapshotFeature.computeDiffBetweenSnapshots()` 方法，这个方法会首先从 `DirectoryWithSnapshotFeature` 的 `DirectoryDiffList` 中取出两个 `Snapshot` 状态间的所有 `ChildrenDiff` 对象，然后调用 `Diff.combinePosterior()` 方法将所有的 `ChildrenDiff` 对象合并为一个 `ChildrenDiff` 对象 `diff`，这个 `diff` 对象汇总了两个快照版本之间进行的所有操作。

```
private void computeDiffRecursively(final INodeDirectory snapshotRoot,
    INode node, List<byte[]> parentPath, SnapshotDiffInfo diffReport) {
    // 获取两个快照的 Snapshot 对象引用
    final Snapshot earlierSnapshot = diffReport.isFromEarlier() ?
        diffReport.getFrom() : diffReport.getTo();
    final Snapshot laterSnapshot = diffReport.isFromEarlier() ?
        diffReport.getTo() : diffReport.getFrom();
    byte[][] relativePath = parentPath.toArray(new byte[parentPath.size()][]); // 当前路径
    if (node.isDirectory()) { // 如果当前节点是目录
        final ChildrenDiff diff = new ChildrenDiff(); // diff 用来保存两个快照间的操作
        INodeDirectory dir = node.asDirectory();
        DirectoryWithSnapshotFeature sf = dir.getDirectoryWithSnapshotFeature();
        if (sf != null) {
            // 调用 computeDiffBetweenSnapshots() 方法合并两个快照间的 diff
            boolean change = sf.computeDiffBetweenSnapshots(earlierSnapshot,
                laterSnapshot, diff, dir);
            if (change) {
                // 如果当前目录在两个快照版本间发生了变化，则写入 diffReport
                diffReport.addDirDiff(dir, relativePath, diff);
            }
        }
        // 调用 getChildrenList() 获取 earlierSnapshot 快照建立时的当前目录的所有子目录项
        ReadOnlyList<INode> children = dir.getChildrenList(earlierSnapshot
            .getId());
        for (INode child : children) { // 遍历所有的子目录项
            final byte[] name = child.getLocalNameBytes();
            boolean toProcess = diff.searchIndex(ListType.DELETED, name) < 0;
            // 如果该子目录项在 laterSnapshot 中已经被删除，则不用递归处理
            if (!toProcess && child instanceof INodeReference.WithName) {
                // 如果发生了 rename 操作，则在 diffReport 中记录 rename 操作
                byte[][] renameTargetPath = findRenameTargetPath(
                    snapshotRoot, (WithName) child,
                    laterSnapshot == null ? Snapshot.CURRENT_STATE_ID :
                        laterSnapshot.getId());
                if (renameTargetPath != null) {
                    toProcess = true;
                    diffReport.setRenameTarget(child.getId(), renameTargetPath);
                }
            }
            // 否则递归处理该子目录项
            if (toProcess) {
                parentPath.add(name);
                computeDiffRecursively(snapshotRoot, child, parentPath, diffReport);
            }
        }
    }
}
```

```

        parentPath.remove(parentPath.size() - 1);
    }
}
else if (node.isFile() && node.asFile().isWithSnapshot()) {
    // 对于文件, 如果发生改变, 也记录在 diffReport 中
    INodeFile file = node.asFile();
    boolean change = file.getFileWithSnapshotFeature()
        .changedBetweenSnapshots(file, earlierSnapshot, laterSnapshot);
    if (change) {
        diffReport.addFileDiff(file, relativePath);
    }
}
}
}

```

(2) 快照流程的代码实现

了解了实现快照功能的类之后, 我们再来学习快照相关流程的代码实现, 包括: 启用目录的快照功能、添加一个快照、在有快照的目录下添加和删除文件, 以及查询快照中文件等流程。

启用快照功能

用户对 HDFS 目录创建快照前, 需要首先启用目标目录的快照功能 (通过执行 “`hdfs dfsadmin-allowSnapshot`” 命令)。只有启用快照功能后, 用户才可以手动在目标目录下创建一个快照 (通过执行 “`hdfs dfs-createSnapshot`” 命令)。

启用快照功能是通过在目标目录对应的 `INodeDirectory` 对象上添加一个 `DirectorySnapshottableFeature` 特性对象实现的, 由 `INodeDirectory.addSnapshottableFeature()` 方法执行具体逻辑。`addSnapshottableFeature()` 方法会首先构造 `DirectorySnapshottableFeature` 对象, 然后调用 `addFeature()` 方法 (参考 `InodeWithAdditionalFileds` 类小节) 将 `Feature` 对象加入 `INodeDirectory` 的 `features` 字段中保存。`addSnapshottableFeature()` 方法的代码如下:

```

public void addSnapshottableFeature() {
    DirectoryWithSnapshotFeature s = this.getDirectoryWithSnapshotFeature();
    // 构造 DirectorySnapshottableFeature 特性
    final DirectorySnapshottableFeature snapshottable =
        new DirectorySnapshottableFeature(s);
    if (s != null) {
        this.removeFeature(s);
    }
    // 将 DirectorySnapshottableFeature 特性添加至 INodeDirectory 的特性列表中
    this.addFeature(snapshottable);
}

```

添加一个快照

开启目录的快照功能后, HDFS 管理员就可以通过执行 “`hdfs dfs-createSnapshot`” 命令添加一个快照了。添加快照的流程是由 `INodeDirectory.addSnapshot()` 方法执行的。

`INodeDirectory.addSnapshot()`方法会首先获取当前目录的 `DirectorySnapshottableFeature` 对象，然后在这个对象上调用 `addSnapshot()`方法添加一个快照到当前目录。

```
public Snapshot addSnapshot(int id, String name) throws SnapshotException,
    QuotaExceededException {
    // 调用 DirectorySnapshottableFeature.addSnapshot() 方法添加一个快照到当前目录
    return getDirectorySnapshottableFeature().addSnapshot(this, id, name);
}
```

`DirectorySnapshottableFeature.addSnapshot()`方法会首先判断当前目录创建的快照数目是否超过了配额，然后构建 `Snapshot` 对象放入 `DirectorySnapshottableFeature.snapshotsByNames` 集合中，最后构建当前快照对应的 `DirectoryDiff` 对象放入 `DirectoryDiffList` 中。这里的 `Snapshot` 对象是用来描述快照信息的，主要保存快照 ID、快照的目录以及快照名等信息。

```
public Snapshot addSnapshot(INodeDirectory snapshotRoot, int id, String name)
    throws SnapshotException, QuotaExceededException {
    // 检查目录的快照数量是否超出了配额
    final int n = getNumSnapshots();
    if (n + 1 > snapshotQuota) {
        throw new SnapshotException("...");
    }
    // 创建 Snapshot 对象
    final Snapshot s = new Snapshot(id, name, snapshotRoot);
    final byte[] nameBytes = s.getRoot().getLocalNameBytes();
    final int i = searchSnapshot(nameBytes);
    if (i >= 0) {
        throw new SnapshotException("..."); // 已经存在同名的快照，抛出异常
    }

    // 创建当前快照对应的 DirectoryDiff 对象，并将该对象放入 DirectoryDiffList 中
    final DirectoryDiff d = getDiffs().addDiff(id, snapshotRoot);
    d.setSnapshotRoot(s.getRoot());
    // 将 Snapshot 对象放入 snapshotsByNames 集合中
    snapshotsByNames.add(-i - 1, s);

    // 更新 modificationTime
    final long now = Time.now();
    snapshotRoot.updateModificationTime(now, Snapshot.CURRENT_STATE_ID);
    s.getRoot().setModificationTime(now, Snapshot.CURRENT_STATE_ID);
    return s;
}
```

添加和删除文件

建立快照之后，向指定的 `INodeDirectory` 中添加和删除文件就不再是简单修改 `INodeDirectory.children` 字段了，这里我们以 `INodeDirectory.addChild()`方法为例，讲解快照建立之后的 `INodeDirectory` 是如何添加和删除子目录项的。

`addChild()`方法首先判断添加的子目录项是否已经在当前的 `INodeDirectory` 中，然后判断

当前 `INodeDirectory` 对象是否在最新的快照中, 如果是则获取 `DirectoryWithSnapshotFeature` 对象, 然后在这个对象上调用 `addChild()` 方法 (参考 `DirectoryWithSnapshotFeature` 类小节)。如果不在最新的快照中, 则调用朴素的 `addChild()` 方法, 直接将 `INode` 节点加入 `INodeDirectory` 的 `children` 集合中。

```
public boolean addChild(INode node, final boolean setModTime,
    final int latestSnapshotId) throws QuotaExceededException {
    final int low = searchChildren(node.getLocalNameBytes());
    if (low >= 0) {
        return false; // 如果要添加的 INode 节点已经在目录中, 则返回
    }

    if (isInLatestSnapshot(latestSnapshotId)) {
        // 获取 DirectoryWithSnapshotFeature 对象
        DirectoryWithSnapshotFeature sf = this.getDirectoryWithSnapshotFeature();
        if (sf == null) {
            sf = this.addSnapshotFeature(null);
        }
        // 在 DirectoryWithSnapshotFeature 上调用 addChild() 方法
        return sf.addChild(this, node, setModTime, latestSnapshotId);
    }
    // 将要添加的 INode 加入 children 字段中保存
    addChild(node, low);
    if (setModTime) {
        // 更新父目录的修改时间
        updateModificationTime(node.getModificationTime(), latestSnapshotId);
    }
    return true;
}

public boolean addChild(INode node) {
    // 查找 INode 节点的插入位置
    final int low = searchChildren(node.getLocalNameBytes());
    if (low >= 0) {
        return false;
    }
    // 调用 addChild() 方法插入 INode 到 children 字段中保存
    addChild(node, low);
    return true;
}

private void addChild(final INode node, final int insertionPoint) {
    if (children == null) {
        children = new ArrayList<INode>(DEFAULT_FILES_PER_DIRECTORY);
    }
    node.setParent(this);
    // 插入 INode 到 children 集合的指定位置
    children.add(-insertionPoint - 1, node);
}
```

```
if (node.getGroupName() == null) {
    node.setGroup(getGroupName());
}
}
```

查询文件

在目录上添加了快照功能之后，获取 `INodeDirectory` 的子目录项就变得复杂了，也就是获取目录在指定快照版本中的某个子目录项。我们参考 `INodeDirectory.getChild()` 方法的实现，`getChild()` 方法首先获取目标目录的 `DirectoryWithSnapshotFeature` 对象，然后调用 `DirectoryWithSnapshotFeature.getChild()` 方法获取指定快照版本中的子目录项。`DirectoryWithSnapshotFeature.getChild()` 方法我们在 `DirectoryWithSnapshotFeature` 类小节中已经介绍过了，请读者参考。

```
public INode getChild(byte[] name, int snapshotId) {
    DirectoryWithSnapshotFeature sf;
    if (snapshotId == Snapshot.CURRENT_STATE_ID ||
        (sf = getDirectoryWithSnapshotFeature()) == null) {
        ReadOnlyList<INode> c = getCurrentChildrenList();
        final int i = ReadOnlyList.Util.binarySearch(c, name);
        return i < 0 ? null : c.get(i);
    }

    return sf.getChild(this, name, snapshotId);
}
```

至此，HDFS 新引入的快照功能就介绍完了。下面我们学习构建（UnderConstruction）特性的实现。

2. FileUnderConstructionFeature 实现

`FileUnderConstructionFeature` 的字面意思就是处于构建状态的特性，构建状态描述的是当客户端为写或者追加写（append）数据打开 HDFS 文件时，文件所处的状态就是构建状态。在 HDFS 2.6 版本之前，使用 `INodeFileUnderConstruction` 类描述处于构建状态的文件；2.6 版本添加 Feature 体系后，则使用 `FileUnderConstructionFeature` 类描述处于构建状态的文件。当客户端打开一个文件进行写或者追加写操作前，会首先调用 `INodeFile.toUnderConstruction()` 方法将该文件转变为构建状态。我们首先看一下 `toUnderConstruction()` 方法的实现。

为了保留原有接口的兼容性，HDFS 并没有将这个方法的命名为 `add*Feature()` 这种形式，而是保留了 `toUnderConstruction()` 这个方法名。`toUnderConstruction()` 方法的实现与 `SnapshotFeature` 是类似的，它首先构造 `FileUnderConstructionFeature` 对象，然后调用 `addFeature()` 将这个 Feature 对象添加到当前的 `INode` 节点上，最后返回 `INodeFile` 的引用。同理，如果将文件从构建状态转变为正常状态，则只需将 `FileUnderConstructionFeature` 对象从 `INode` 中移除即可，对应于方法 `toCompleteFile()`。`toUnderConstruction()` 方法的代码如下：

```
INodeFile toUnderConstruction(String clientName, String clientMachine) {
    Preconditions.checkNotNull(!isUnderConstruction(),
```

```

        "file is already under construction");
    // 构造 FileUnderConstructionFeature 对象
    FileUnderConstructionFeature uc = new FileUnderConstructionFeature(
        clientName, clientMachine);
    // 添加到 INode 的 features 字段中保存
    addFeature(uc);
    return this;
}

```

HDFS 写文件和追加写（append）文件的流程是非常复杂的，所以 FileUnderConstructionFeature 中需要记录一些与写操作相关的属性，包括：

- **clientName**: 发起文件写操作的客户端名称，这个属性也用于租约管理功能。在 HDFS 中，租约是名字节点维护的给予客户端在一定期限内可以进行文件写操作的权限的合同，我们会在租约管理小节中介绍。
- **clientMachine**: 客户端所在主机。

如图 3-16 所示，FileUnderConstructionFeature 定义的方法包括了 clientName 和 clientMachine 两个属性的 get/set 方法，updateLengthOfLastBlock()方法用于更新文件最后一个正在写的数据块的长度，cleanZeroSizeBlock()方法则用于删除快照中文件的最后一个长度为 0 的数据块。

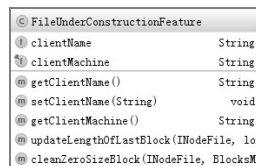


图 3-16 FileUnderConstructionFeature 类结构图

3.1.3 FSEditLog 类

在 Namenode 中，命名空间（namespace，指文件系统目录树、文件元数据等信息）是被全部缓存在内存中的，一旦 Namenode 重启或者宕机，内存中的所有数据将会全部丢失，所以必须要有一种机制能够将整个命名空间持久化保存，并且能在 Namenode 重启时重建命名空间。

目前 Namenode 的实现是将命名空间信息记录在一个叫作 fsimage（命名空间镜像）的二进制文件中，fsimage 将文件系统目录树中的每个文件或者目录的信息保存为一条记录，每条记录中包括了该文件（或目录）的名称、大小、用户、用户组、修改时间、创建时间等信息。Namenode 重启时，会读取这个 fsimage 文件来重构命名空间。但是 fsimage 始终是磁盘上的一个文件，不可能时时刻刻都跟 Namenode 内存中的数据保持同步，并且 fsimage 文件一般都很大（GB 级别的很常见），如果所有的更新操作都实时地写入 fsimage 文件，则会导致 Namenode 运行得十分缓慢，所以 HDFS 每过一段时间才更新一次 fsimage 文件。

那么问题来了，在一个新 `fsimage` 文件和上一个 `fsimage` 文件中进行的 `Namenode` 操作保存在哪里呢？HDFS 将这些操作记录在 `editlog`（编辑日志）文件中，`editlog` 是一个日志文件，HDFS 客户端执行的所有写操作首先会被记录到 `editlog` 文件中。HDFS 会定期地将 `editlog` 文件与 `fsimage` 文件进行合并，以保持 `fsimage` 跟 `Namenode` 内存中记录的命名空间完全同步。本节和下一节就重点介绍 `editlog` 以及 `fsimage` 的实现。

在 HDFS 源码中，使用 `FSEditLog` 类来管理 `editlog` 文件。和 `fsimage` 文件不同，`editlog` 文件会随着 `Namenode` 的运行实时更新，所以 `FSEditLog` 类的实现依赖于底层的输入流和输出流，同时 `FSEditLog` 类还需要对外提供大量的 `log*()` 方法用于记录命名空间的修改操作。本节首先介绍 `FSEditLog` 类使用到的 `transactionId` 机制，然后介绍 `FSEditLog` 类的状态机以及对外提供的方法，最后介绍 `editlog` 文件的输入流和输出流。

1. transactionId 机制

HDFS 的 `editlog` 文件可以存放在多种容器中，比如文件系统（`FileJournalManager` 类管理）、共享 NFS（`BackupJournalManager` 类管理）、Bookkeeper（`BookkeeperJournalManager` 类管理）等。而管理这些不同容器内文件的方法也有很多种，目前 HDFS 采用的是基于 `transactionId` 的日志管理方法。

这里我们以 `Namenode` 元数据文件夹为例。图 3-17 显示的是在 `Namenode` 元数据文件夹中运行 `tree` 命令的结果，`Namenode` 元数据文件夹是存放 `fsimage` 文件和 `editlog` 文件的文件夹，由 `hdfs-site.xml` 配置文件的 `dfs.namenode.name.dir` 配置项配置。在这个示例中，元数据文件夹是 `data/dfs/name` 目录。

```
data/dfs/name
├── current
│   └── VERSION
├── edits_00000000000000000001-000000000000000007
├── edits_00000000000000000008-000000000000000015
├── edits_00000000000000000016-000000000000000022
├── edits_00000000000000000023-000000000000000029
├── edits_00000000000000000030-000000000000000030
├── edits_00000000000000000031-000000000000000031
├── edits_inprogress_00000000000000000032
├── fsimage_00000000000000000030
├── fsimage_00000000000000000030.md5
├── fsimage_00000000000000000031
├── fsimage_00000000000000000031.md5
├── seen_txid
└── in_use.lock
```

图 3-17 Namenode 元数据文件夹结构

这里我们首先了解一下 `transactionId` 的概念。`TransactionId` 与客户端每次发起的 `RPC` 操作相关，当客户端发起一次 `RPC` 请求对 `Namenode` 的命名空间修改后，`Namenode` 就会在 `editlog` 中发起一个新的 `transaction` 用于记录这次操作，每个 `transaction` 会用一个唯一的 `transactionId` 标识。理解清楚了 `transactionId` 的概念之后，我们看一下这个文件夹中每个文件都有什么作用。

- `edits_start transaction id-end transaction id`: `edits` 文件就是我们描述的 `editlog` 文件，`edits` 文件中存放的是客户端执行的所有更新命名空间的操作。每个 `edits` 文件都包含了文件名中 `start transaction id - end transaction id` 之间的所有事务。这里以 `edits_000*01 -`

000*07 为例, 这个文件记录了 transaction id 在 1 和 7 之间的所有事务 (transaction)。

- **edits_inprogress_start transaction id:** 正在进行处理的 editlog。所有从 start transaction id 开始的新的修改操作都会记录在这个文件中, 直到 HDFS 重置 (roll) 这个日志文件。重置操作会将 inprogress 文件关闭, 并将 inprogress 文件改名为正常的 editlog 文件(如上一项所示), 同时还会打开一个新的 inprogress 文件, 记录正在进行的事务。例如当前文件夹中的 edits_inprogress_0000000000000000032 文件, 这个文件记录了所有 transaction id 大于 32 的新开始的事务, 我们将这个事务区间称为一个日志段落 (segment)。Namenode 元数据文件夹中存在这个文件有两种可能, 要么是 Active Namenode 正在写入数据, 要么是前一个 Namenode 没有正确地关闭。
- **fsimage_end transaction id:** fsimage 文件是 Hadoop 文件系统元数据的一个永久性的检查点, 包含 Hadoop 文件系统中 end transaction id 前的完整的 HDFS 命名空间元数据镜像, 也就是 HDFS 所有目录和文件对应的 INode 的序列化信息。以当前文件夹为例, fsimage_000*31 就是 fsimage_000*30 与 edits_000*31-000*31 合并后的镜像文件, 保存了 transaction id 小于 32 的 HDFS 命名空间的元数据。每个 fsimage 文件还有一个对应的 md5 文件, 用来确保 fsimage 文件的正确性, 以防止磁盘异常发生。
- **seen_txid:** 这个文件中保存了上一个检查点 (checkpoint) (合并 edits 和 fsimage 文件) 以及编辑日志重置 (editlog roll) (持久化当前的 inprogress 文件并且创建一个新的 inprogress 文件) 时最新的事务 id (transaction id)。要特别注意的是, 这个事务 id 并不是 Namenode 内存中最新的事务 id, 因为 seen_txid 只在检查点操作以及编辑日志重置操作时更新。这个文件的作用在于 Namenode 启动时, 可以利用这个文件判断是否有 edits 文件丢失。例如, Namenode 使用不同的目录保存 fsimage 以及 edits 文件, 如果保存 edits 的目录内容丢失, Namenode 将会使用上一个检查点保存的 fsimage 启动, 那么上一个检查点之后的所有事务都会丢失。为了防止发生这种状况, Namenode 启动时会检查 seen_txid 并确保内存中加载的事务 id 至少超过 seen_txid; 否则 Namenode 将终止启动操作。

2. FSEditLog 状态机

介绍完 transactionId 机制后, 我们来学习 FSEditLog 类的实现。FSEditLog 类被设计成一个状态机, 用内部类 FSEditLog.State 描述。FSEditLog 有以下 5 个状态。

- **UNINITIALIZED:** editlog 的初始状态。
- **BETWEEN_LOG_SEGMENTS:** editlog 的前一个 segment 已经关闭, 新的还没开始。
- **IN_SEGMENT:** editlog 处于可写状态。
- **OPEN_FOR_READING:** editlog 处于可读状态。
- **CLOSED:** editlog 处于关闭状态。

对于非 HA 机制的情况, FSEditLog 应该开始于 UNINITIALIZED 或者 CLOSED 状态 (因为在构造 FSEditLog 对象时, FSEditLog 的成员变量 state 默认为 State.UNINITIALIZED)。FSEditLog 初始化完成之后进入 BETWEEN_LOG_SEGMENTS 状态, 表示前一个 segment 已经关闭, 新的还没开始, 日志已经做好了准备。当打开日志服务时, 改变 FSEditLog 状态为

IN_SEGMENT 状态，表示可以写 editlog 文件了。

对于 HA 机制的情况，FSEditLog 同样应该开始于 UNINITIALIZED 或者 CLOSED 状态，但是在完成初始化后 FSEditLog 并不进入 BETWEEN_LOG_SEGMENTS 状态，而是进入 OPEN_FOR_READING 状态（因为目前 Namenode 启动时都是以 Standby 模式启动的，然后通过 DFSHAAdmin 发送命令把其中一个 Standby NameNode 转换成 Active Namenode）。

下面我们看一下 FSEditLog 的状态转移图，如图 3-18 所示。图中所示的方法会在后面的小节中详细介绍。

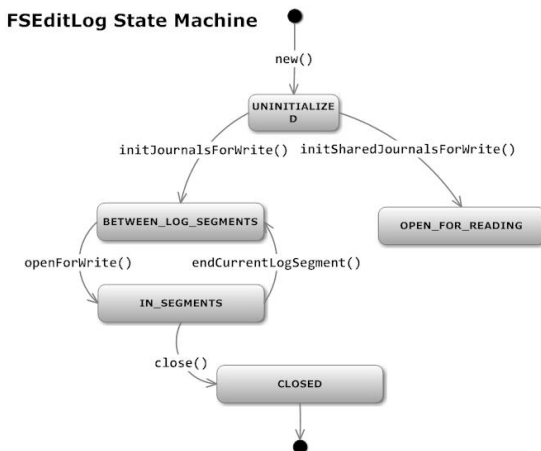


图 3-18 FSEditLog 的状态转移图

(1) initJournalsForWrite()

initJournalsForWrite()方法是 FSEditLog 的 public 方法，调用这个方法会将 FSEditLog 从 UNINITIALIZED 状态转换为 BETWEEN_LOG_SEGMENTS 状态。

```
public synchronized void initJournalsForWrite() {
    Preconditions.checkState(state == State.UNINITIALIZED ||
        state == State.CLOSED, "Unexpected state: %s", state);
    // 检查之前的状态
    initJournals(this.editsDirs); // 调用 initJournals() 方法
    state = State.BETWEEN_LOG_SEGMENTS; // 状态转换为 BETWEEN_LOG_SEGMENTS
}
```

initJournalsForWrite()方法调用了 initJournals()方法，initJournals()方法会根据传入的 dirs 变量（保存的是 editlog 文件的存储位置，都是 URI）初始化 journalSet 字段（JournalManager 对象的集合）。初始化之后，FSEditLog 就可以调用 journalSet 对象的方法向多个日志存储位置写 editlog 文件了。

JournalManager 类是负责在特定存储目录上持久化 editlog 文件的类，它的 format()方法负责格式化底层存储，startLogSegment()方法负责从指定事务 id 开始记录一个操作的段落，

finalizeLogSegment()方法负责完成指定事务 id 区间的写操作。这里之所以抽象这个接口，是因为 Namenode 可能将 editlog 文件持久化到不同类型的存储上，也就需要不同类型的 JournalManager 来管理，所以需要定义一个抽象的接口。如图 3-19 所示，JournalManager 有多个子类，普通的文件系统由 FileJournalManager 类管理、共享 NFS 由 BackupJournalManager 类管理、Bookkeeper 由 BookkeeperJournalManager 类管理、Quorum 集群则由 QuorumJournalManager 类管理。

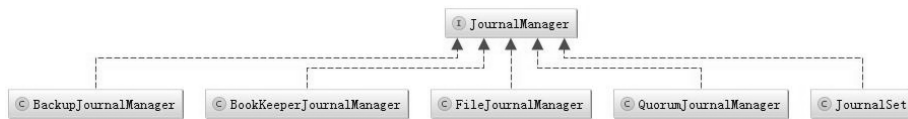


图 3-19 JournalManager 继承关系图

了解了 JournalManager 后，我们看一下 initJournals()方法的实现，代码如下：

```

private synchronized void initJournals(List<URI> dirs) {
    int minimumRedundantJournals = conf.getInt(
        DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_KEY,
        DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_DEFAULT);

    // 初始化 journalSet 集合，存放存储路径对应的所有 JournalManager 对象
    journalSet = new JournalSet(minimumRedundantJournals);

    // 根据传入的 URI 获取对应的 JournalManager 对象
    for (URI u : dirs) {
        boolean required = FSNamesystem.getRequiredNamespaceEditsDirs(conf)
            .contains(u);
        if (u.getScheme().equals(NNStorage.LOCAL_URI_SCHEME)) {
            StorageDirectory sd = storage.getStorageDirectory(u);
            if (sd != null) {
                // 本地 URI，则加入 FileJournalManager 即可
                journalSet.add(new FileJournalManager(conf, sd, storage), required);
            }
        } else {
            // 否则根据 URI 创建对应的 JournalManager 对象，并放入 journalSet 中保存
            journalSet.add(createJournal(u), required);
        }
    }
}

```

(2) initSharedJournalsForRead()

initSharedJournalsForRead()方法是 FSEditLog 的 public 方法，用在 HA 情况下。调用这个方法会将 FSEditLog 从 UNINITIALIZED 状态转换为 OPEN_FOR_READING 状态。

与 initJournalsForWrite()方法相同，initSharedJournalsForRead()方法也调用了 initJournals()方法执行初始化操作，只不过 editlog 文件的存储位置不同，在 HA 的情况下，editlog 文件的

存储目录为共享存储目录，这个共享存储目录由 Active Namenode 和 Standby Namenode 共享读取。initSharedJournalsForRead()方法的代码如下：

```
public synchronized void initSharedJournalsForRead() {
    if (state == State.OPEN_FOR_READING) {
        LOG.warn("Initializing shared journals for READ, already open for READ",
            new Exception());
        return;
    }
    Preconditions.checkState(state == State.UNINITIALIZED ||
        state == State.CLOSED);
    // 对于 HA 的情况，editlog 的日志存储目录为共享的目录 sharedEditsDirs
    initJournals(this.sharedEditsDirs);
    state = State.OPEN_FOR_READING;
}
```

(3) openForWrite()

openForWrite()方法用于初始化 editlog 文件的输出流，并且打开第一个日志段落（log segment）。在非 HA 机制下，调用这个方法会完成 BETWEEN_LOG_SEGMENTS 状态到 IN_SEGMENT 状态的转换。本节结合下面的目录结构说明 openForWrite()方法的实现。

```
data/dfs/name
├─ current
│   └─ VERSION
│   └─ edits_00000000000000000001-0000000000000000007
│   └─ edits_00000000000000000008-0000000000000000015
│   └─ edits_00000000000000000016-0000000000000000022
│   └─ edits_00000000000000000023-0000000000000000029
│   └─ edits_00000000000000000030-0000000000000000030
│   └─ edits_00000000000000000031-0000000000000000031
│   └─ fsimage_00000000000000000030
│   └─ fsimage_00000000000000000030.md5
│   └─ fsimage_00000000000000000031
│   └─ fsimage_00000000000000000031.md5
│   └─ seen_txid
└─ in_use.lock
```

如图 3-20 所示，openForWrite()方法的执行分为这样几个部分，分别调用了三个不同的方法。

- getLastWrittenTxId()——查找已经写到 editlog 日志文件中的最新的 transactionId，对应图 3-20 中的情况，返回的是 31。
- journalSet.selectInputStreams()——传入了参数 segmentTxId，这个参数会作为这次操作的 transactionId，值为 editlog 已经记录的最新的 transactionId 加 1（这里是 31+1=32）。selectInputStreams()方法会判断有没有一个以 segmentTxId（32）开始的日志（从上面的例子来看，是没有的），如果没有则表示当前 transactionId 的值选择正确，可以打开新的 editlog 文件记录以 segmentTxId 开始的日志段落。如果方法找到了包含这

个 transactionId 的 editlog 文件,则表示出现了两个日志 transactionId 交叉的情况,抛出异常。

- startLogSegment()——开始记录 transactionId 为 32 的日志段落,新建 edits_inprogress_32 文件。同时将 FSEditlog 的状态转变为 IN_SEGMENT。

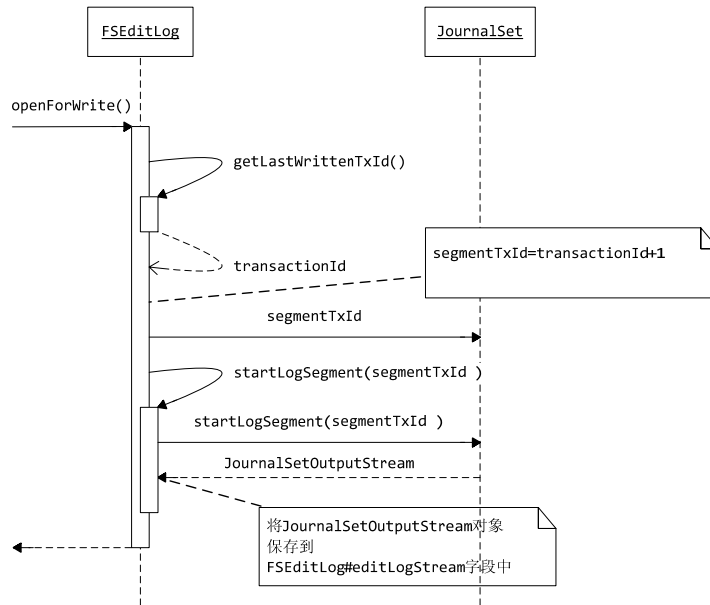


图 3-20 openForWrite()方法调用流程

openForWrite()方法的代码如下:

```

synchronized void openForWrite() throws IOException {
    Preconditions.checkState(state == State.BETWEEN_LOG_SEGMENTS,
        "Bad state: %s", state);

    // 返回最后一个写入 log 的 transactionId+1, 作为本次操作的 transactionId
    long segmentTxId = getLastWrittenTxId() + 1;

    // 这里判断, 有没有包含这个新的 segmentTxId 的 editlog 文件, 如果有则抛出异常
    List<EditLogInputStream> streams = new ArrayList<EditLogInputStream>();
    journalSet.selectInputStreams(streams, segmentTxId, true, true);
    if (!streams.isEmpty()) {
        // ...
        throw new IllegalStateException(error);
    }

    // 调用 startLogSegment() 方法
  }

```

```
startLogSegment(segmentTxId, true);
assert state == State.IN_SEGMENT : "Bad state: " + state;
}
```

下面我们看看 `startLogSegment()` 方法的实现。这个方法调用了 `journalSet.startLogSegment()` 方法在所有 `editlog` 文件的存储路径上构造输出流，并将这些输出流保存在 `FSEditLog` 的字段 `journalSet.journals` 中。`journalSet` 的 `journals` 字段是一个 `JournalAndStream` 对象的集合，这个集合中的每一个 `JournalAndStream` 对象都封装了一个 `JournalManager`，以及这个 `JournalManager` 打开的 `editlog` 文件的输出流，那么 `journals` 字段就保存了 `editlog` 文件在所有存储路径上的输出流。`startLogSegment()` 方法会构造一个 `JournalSetOutputStream` 对象，并将这个对象保存在 `FSEditLog` 的 `editLogStream` 字段中，`FSEditLog` 之后进行的所有写操作都是通过 `editLogStream` 引用的 `JournalSetOutputStream` 对象进行的。`JournalSetOutputStream` 类是 `EditLogOutputStream` 的子类，在 `JournalSetOutputStream` 对象上调用的所有接口方法都会被前转到 `journalSet.journals` 字段中保存的 `editlog` 文件在所有存储路径上的输出流对象上（通过调用 `mapJournalsAndReportErrors()` 方法实现，请参考 `JournalSetOutputStream` 小节）。`JournalSetOutputStream` 类的实现我们会在 `EditLogOutputStream` 小节中介绍。`journalSet` 就是通过这种方式，将多个存储位置上的输出流对外封装成了一个输出流，大大方便了调用。`startLogSegment()` 方法最后会将 `FSEditlog.curSegmentTxId` 字段（`FSEditlog` 当前正在写入 `txid`）设置为传入的 `segmentTxId`，同时将 `editlog` 的状态更改为 `IN_SEGMENT` 状态。`startLogSegment()` 方法的代码如下：

```
synchronized void startLogSegment(final long segmentTxId,
    boolean writeHeaderTxn) throws IOException {
    // 检查条件代码，省略
    // ...

    // 初始化 editLogStream
    try {
        editLogStream = journalSet.startLogSegment(segmentTxId);
    } catch (IOException ex) {
        throw new IOException("Unable to start log segment " +
            segmentTxId + ": too few journals successfully started.", ex);
    }

    // 当前正在写入 txid 设置为 segmentTxId
    curSegmentTxId = segmentTxId;
    state = State.IN_SEGMENT;
    //...
}
```

(4) `endCurrentLogSegment()`

`endCurrentLogSegment()` 会将当前正在写入的日志段落关闭，它调用了 `journalSet.finalizeLogSegment()` 方法将 `curSegmentTxId` -> `lastTxId` 之间的操作持久化到磁盘上。如上例中，调用 `endCurrentLogSegment()` 方法就会产生 `editlog` 文件 `edits_0032-0034`。同时这个方法会将

FSEditLog 状态机更改为 BETWEEN_LOG_SEGMENTS 状态。

```
synchronized void endCurrentLogSegment(boolean writeEndTxn) {
    LOG.info("Ending log segment " + curSegmentTxId);
    Preconditions.checkState(isSegmentOpen(),
        "Bad state: %s", state);

    // ...
    // 获取当前写入的最后一个 id
    final long lastTxId = getLastWrittenTxId();

    try {
        // 调用 journalSet.finalizeLogSegment 将 curSegmentTxid -> lastTxId 之间的操作
        // 写入磁盘 (例如 editlog 文件 edits_0032-0034)
        journalSet.finalizeLogSegment(curSegmentTxId, lastTxId);
        editLogStream = null;
    } catch (IOException e) {
    }
    // 更改状态机的状态
    state = State.BETWEEN_LOG_SEGMENTS;
}
```

journalSet.finalizeLogSegment() 方法也会调用 mapJournalsAndReportErrors() 方法将 finalizeLogSegment() 调用前转到 journals 集合中保存的所有的 JournalManager 对象上。这里我们以 FileJournalManager 为例, FileJournalManager.finalizeLogSegment() 方法会将 edit_inprogress 文件改名为 edit 文件, 新生成的 edit 文件覆盖了 curSegmentTxid -> lastTxId 之间的所有事务。FileJournalManager.finalizeLogSegment() 方法的代码如下:

```
synchronized public void finalizeLogSegment(long firstTxId, long lastTxId)
    throws IOException {
    File inprogressFile = NNStorage.getInProgressEditsFile(sd, firstTxId); // 原有的 inprogress 文件
    File dstFile = NNStorage.getFinalizedEditsFile(
        sd, firstTxId, lastTxId); // 构造新的 edit 文件

    try {
        NativeIO.renameTo(inprogressFile, dstFile); // 重命名 edit 文件
    } catch (IOException e) {
        errorReporter.reportErrorOnFile(dstFile);
        throw new IllegalStateException("Unable to finalize edits file " + inprogressFile, e);
    }

    // ...
}
```

(5) close()

close() 方法用于关闭 editlog 文件的存储, 完成了 IN_SEGMENT 到 CLOSED 状态的改变。

close()会首先等待 sync 操作完成, 然后调用上一节介绍的 endCurrentLogSegment()方法, 将当前正在进行写操作的日志段落结束。之后 close()方法会关闭 journalSet 对象, 并将 FSEditLog 状态机转变为 CLOSED 状态。这里提到的 sync 操作我们将在后面的小节中介绍。close()方法的代码如下:

```
synchronized void close() {
    try {
        if (state == State.IN_SEGMENT) {
            assert editLogStream != null;
            // 如果有 sync 操作, 则等待 sync 操作完成
            waitForSyncToFinish();
            // 结束当前 logSegment
            endCurrentLogSegment(true);
        }
    } finally {
        // 关闭 journalSet
        if (journalSet != null && !journalSet.isEmpty()) {
            try {
                journalSet.close();
            } catch (IOException ioe) {
                LOG.warn("Error closing journalSet", ioe);
            }
        }
        // 将状态机更改为 CLOSED 状态
        state = State.CLOSED;
    }
}
```

3. EditLogOutputStream

FSEditLog 类会调用 FSEditLog.editLogStream 字段的 write()方法在 editlog 文件中记录一个操作, 数据会先被写入到 editlog 文件输出流的缓存中, 然后 FSEditLog 类会调用 editLogStream.flush()方法将缓存中的数据同步到磁盘上。FSEditLog 的 editLogStream 字段是 EditLogOutputStream 类型的, EditLogOutputStream 类是一个抽象类, 它定义了向持久化存储上写 editlog 文件的相关接口。

由于目前 Namenode 可以在多种类型的异构存储上保存 editlog 文件, 例如普通文件系统、共享 NFS、Bookkeeper 以及 Quorum 集群等, 所以 EditLogOutputStream 定义了多个子类来向不同存储系统上的 editlog 文件中写入数据。EditLogOutputStream 的类图如图 3-21 所示, EditLogFileOutputStream 抽象了本地文件系统上 editlog 文件的输出流, BookKeeperEditLogOutputStream 抽象了 BookKeeper 系统上 editlog 文件的输出流, QuorumOutputStream 抽象了 Quorum 集群上 editlog 文件的输出流。同时由于 Namenode 可以同时向多个不同的存储上写入 editlog 文件, 所以 EditLogOutputStream 还定义了子类 JournalSetOutputStream 执行聚合的写入操作。

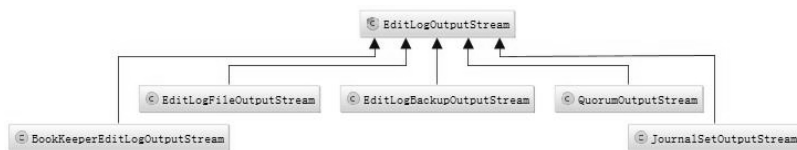


图 3-21 EditLogOutputStream 继承关系图

本节就重点介绍 EditLogOutputStream、JournalSetOutputStream 以及 EditLogFileOutputStream 三个类的实现，其他 EditLogOutputStream 的子类由于篇幅原因就不再介绍了，请读者自行阅读源代码。

(1) JournalSetOutputStream

JournalSetOutputStream 类是 EditLogOutputStream 的子类，在 JournalSetOutputStream 对象上调用的所有 EditLogOutputStream 接口方法都会被前转到 FSEditLog.journalSet 字段中保存的 editlog 文件在所有存储位置上的输出流对象上（通过调用 mapJournalsAndReportErrors() 方法实现）。

FSEditLog 的 editLogStream 字段就是 JournalSetOutputStream 类型的（是在 startLogSegment() 方法中赋值的，请参考 FSEditLog 状态机的 openForWrite() 小节），通过调用 JournalSetOutputStream 对象提供的方法，FSEditLog 可以将 Namenode 多个存储位置上的 editlog 文件输出流对外封装成一个输出流，大大方便了调用。图 3-22 给出了 JournalSetOutputStream 调用流程。

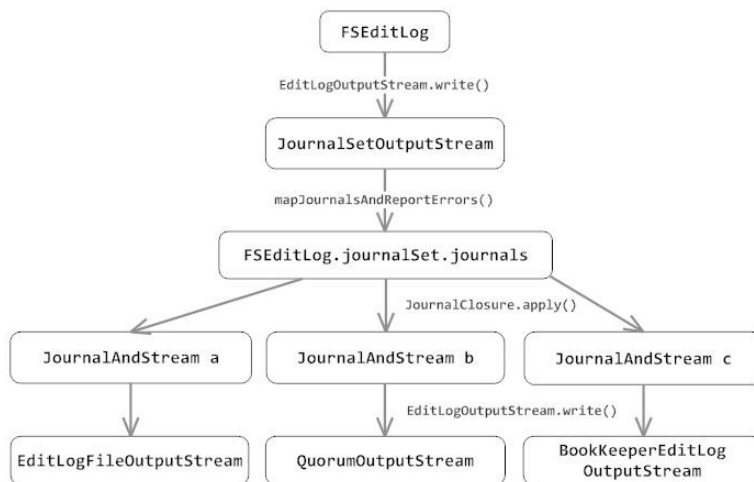


图 3-22 JournalSetOutputStream 调用流程

JournalSetOutputStream 类是通过 mapJournalsAndReportErrors() 方法，将 EditLogOutputStream 接口上的 write() 调用前转到了 FSEditLog 中保存的所有存储路径上 editlog 文件对应的 EditLogOutputStream 输出流对象上的。这个方法会遍历 FSEditLog.journalSet.journals 集合，

然后将 `write()` 请求前转到 `journals` 集合中保存的所有 `JournalAndStream` 对象上。`journalSet` 的 `journals` 字段是一个 `JournalAndStream` 对象的集合, `JournalAndStream` 对象封装了一个 `JournalManager` 对象, 以及在这个 `JournalManager` 上打开的 `editlog` 文件的 `EditLogOutputStream` 对象。`journalSet.journals` 字段是在 `FSEditLog.startLogSegment()` 方法中赋值的 (请参考 `FSEditLog` 状态机的 `openForWrite()` 小节), 这个方法调用了 `journalSet.startLogSegment()` 方法在所有 `editlog` 文件的存储路径上构造输出流, 并将这些输出流保存在 `FSEditLog` 的 `journalSet.journals` 字段中。`mapJournalsAndReportErrors()` 方法的代码如下:

```
private void mapJournalsAndReportErrors(
    JournalClosure closure, String status) throws IOException{

    List<JournalAndStream> badJAS = Lists.newLinkedList();
    // 遍历 journals 字段中保存的所有 JournalAndStream 对象
    for (JournalAndStream jas : journals) {
        try {
            // 在闭包对象上调用 apply() 方法前转请求
            closure.apply(jas);
        } catch (Throwable t) {
            if (jas.isRequired()) {
                abortAllJournals();
                terminate(1, msg);
            } else {
                badJAS.add(jas);
            }
        }
    }
    disableAndReportErrorOnJournals(badJAS);
    if (!NameNodeResourcePolicy.areResourcesAvailable(journals,
        minimumRedundantJournals)) {
        String message = status + " failed for too many journals";
        throw new IOException(message);
    }
}
```

可以看到, `mapJournalsAndReportErrors()` 方法在调用时传入了一个闭包对象 `closure`, 这个对象是在 `JournalSetOutputStream` 实现的 `EditLogOutputStream` 接口方法上定义的。我们以 `JournalSetOutputStream.write()` 方法为例, `write()` 方法定义了写操作的闭包对象, 这个闭包对象会提取出 `JournalAndStream` 对象中封装的 `EditLogOutputStream` 对象, 然后调用这个对象上的 `write()` 方法来完成写数据的功能。通过这种闭包机制, `JournalSetOutputStream` 完成了将 `EditLogOutputStream` 接口上的 `write()` 调用前转到 `JournalAndStream` 保存的 `EditLogOutputStream` 对象上的操作。

```
public void write(final FSEditLogOp op)
    throws IOException {
    mapJournalsAndReportErrors(new JournalClosure() {
        @Override
        public void apply(JournalAndStream jas) throws IOException {
```

```

        if (jas.isActive()) {
            jas.getCurrentStream().write(op); // 提取出 JournalAndStream 对象中封装的
            EditLogOutputStream对象, 并在 EditLogOutputStream对象上调用 write() 方法
        }
    }
    }, "write op");
}

```

`JournalSetOutputStream` 类中其他接口方法的实现也类似于 `write()` 方法, 这种设计模式非常巧妙, 请读者注意积累。

（2）EditLogFileOutputStream

`EditLogFileOutputStream` 是向本地文件系统中保存的 `editlog` 文件写数据的输出流, 向 `EditLogFileOutputStream` 写数据时, 数据首先被写入到输出流的缓冲区中, 当显式地调用 `flush()` 操作后, 数据才会从缓冲区同步到 `editlog` 文件中。

`EditLogFileOutputStream` 提供了一种巧妙的双 `buffer` 模式来缓存输出流数据, 本节先介绍这种模式, 然后介绍 `EditLogFileOutputStream` 的具体实现, 包括它的字段、构造方法以及对外提供的接口方法。

EditsDoubleBuffer 类

数据在写入 `editlog` 文件前, 会首先被写入到输出流的缓冲区中。`EditLogFileOutputStream` 的缓冲区用到了一个比较特殊的数据结构——`EditsDoubleBuffer`, 本节就介绍 `EditsDoubleBuffer` 的实现。

`EditsDoubleBuffer` 中包括两块缓存, 数据会先被写入到 `EditsDoubleBuffer` 的一块缓存中, 而 `EditsDoubleBuffer` 的另一块缓存可能正在进行磁盘的同步操作（就是将缓存中的文件写入磁盘的操作）。`EditsDoubleBuffer` 这样的设计会保证输出流进行磁盘同步操作的同时, 并不影响数据写入的功能。`EditsDoubleBuffer` 定义的字段如下代码所示:

```

private TxnBuffer bufCurrent; // 正在写入的缓冲区
private TxnBuffer bufReady; // 准备好同步的缓冲区
private final int initBufferSize; // 缓冲区的大小

```

输出流要进行同步操作时, 首先要调用 `EditsDoubleBuffer.setReadyToFlush()` 方法交换两个缓冲区, 将正在写入的缓存改变为同步缓存, 然后才可以进行同步操作。`setReadyToFlush()` 方法的代码如下:

```

public void setReadyToFlush() {
    assert isFlushed() : "previous data not flushed yet";
    TxnBuffer tmp = bufReady; // 交换两个缓冲区
    bufReady = bufCurrent;
    bufCurrent = tmp;
}

```

完成了 `setReadyToFlush()` 调用之后, 输出流就可以调用 `flushTo()` 方法将同步缓存中的数

写入到文件中。flushTo()方法的代码如下：

```
public void flushTo(OutputStream out) throws IOException {
    bufReady.writeTo(out); // 将同步缓存中的数据写入文件
    bufReady.reset(); // 将同步缓存中保存的数据清空
}
```

双缓冲区数据结构 EditsDoubleBuffer 将同步操作与写入操作相分离的思想很值得我们学习，请读者注意积累。

字段和构造方法

如图 3-23 所示，EditLogFileOutputStream 定义了多个字段。



EditLogFileOutputStream	
LOG	Log
MIN_PREALLOCATION_LENGTH	int
file	File
fp	FileOutputStream
fc	FileChannel
doubleBuf	EditsDoubleBuffer
fill	ByteBuffer
shouldSyncWritesAndSkipFsync	boolean
shouldSkipFsyncForTests	boolean
write(FSEditLogOp)	void
writeRaw(byte[], int, int)	void
create(int)	void
writeHeader(int, DataOutputStream)	void
close()	void
abort()	void
setReadyToFlush()	void
flushAndSync(boolean)	void
shouldForceSync()	boolean
preallocate()	void
getFile()	File
toString()	String
isOpen()	boolean
setFileChannelForTesting(FileChannel)	void
getFileChannelForTesting()	FileChannel
setShouldSkipFsyncForTesting(boolean)	void

图 3-23 EditLogFileOutputStream 字段

- file: 输出流对应的 editlog 文件。
- fp: editlog 文件对应的输出流。
- fc: editlog 文件对应的输出流通道。
- doubleBuf: 一个具有两块缓存的缓冲区，数据必须先写入缓存，然后再由缓存同步到磁盘上。请参考上一小节中 EditsDoubleBuffer 的实现。
- fill: 用来扩充 editlog 文件大小的数据块。当要进行同步操作时，如果 editlog 文件不够大，则使用 fill 来扩充 editlog。请参考方法小节中 preallocate()方法的实现。

EditLogFileOutputStream 有一个 static 的代码段，将 fill 字段用 FSEditLogOpCodes.OP_INVALID 字节填满。

```
static {
    fill.position(0);
    for (int i = 0; i < fill.capacity(); i++) {
```



```

        fill.put(FSEditLogOpCodes.OP_INVALID.getOpCode());
    }
}

```

上面方法中提到的 `FSEditLogOpCodes` 是一个枚举，对应于 `editlog` 文件中记录的操作的类型，每种情况都使用一个 `byte` 表示。

```

public enum FSEditLogOpCodes {
    OP_INVALID            ((byte) -1),    // 不合法的操作
    OP_ADD                ((byte) 0),     // 添加操作
    OP_RENAME_OLD         ((byte) 1),     // 重命名操作
    //..
}

```

`EditLogFileOutputStream` 的构造方法比较简单，初始化了定义的所有字段。

```

public EditLogFileOutputStream(Configuration conf, File name, int size)
    throws IOException {
    super();
    // ...
    file = name;
    doubleBuf = new EditsDoubleBuffer(size);
    RandomAccessFile rp;
    if (shouldSyncWritesAndSkipFsync) {
        rp = new RandomAccessFile(name, "rws");
    } else {
        rp = new RandomAccessFile(name, "rw");
    }
    fp = new FileOutputStream(rp.getFD());
    fc = rp.getChannel();
    fc.position(fc.size());
}

```

方法

`EditLogFileOutputStream` 的 `write()` 方法、`setReadyToFlush()` 方法分别用于向输出流中写入操作，以及为同步操作做准备，这两个方法的实现都比较简单，直接调用 `doubleBuf` 中的对应方法即可，代码如下：

```

public void write(FSEditLogOp op) throws IOException { // 向输出流写入一个操作
    doubleBuf.writeOp(op); // 向 doubleBuf 写入 FSEditLogOp 对象
}

public void setReadyToFlush() throws IOException { // 为同步数据做准备
    doubleBuf.setReadyToFlush(); // 调用 doubleBuf.setReadyToFlush() 交换两个缓冲区
}

```

`flushAndSync()` 方法则用于将输出流中缓存的数据同步到磁盘上的 `editlog` 文件中。`flushAndSync()` 方法首先调用了 `preallocate()` 方法，`preallocate()` 方法用于在 `editLog` 文件大小不够时，填充 `editlog` 文件。之后 `flushAndSync()` 会调用 `doubleBuf.flushTo()` 方法将缓存中的数据

同步到 editlog 文件中。flushAndSync()方法的代码如下：

```
public void flushAndSync(boolean durable) throws IOException {
    // ..
    preallocate(); // 如果 editlog 文件大小不够，则扩充文件大小
    doubleBuf.flushTo(fp); // 将缓存中的数据刷新到 editlog 文件
    //..
}

private void preallocate() throws IOException {
    long position = fc.position();
    long size = fc.size();
    int bufSize = doubleBuf.getReadyBuf().getLength();
    long need = bufSize - (size - position); // 判断需要扩充容量的大小
    if (need <= 0) {
        return;
    }
    long oldSize = size;
    long total = 0;
    long fillCapacity = fill.capacity();
    while (need > 0) {
        fill.position(0);
        IOUtils.writeFully(fc, fill, size); // 将填充缓冲区写入通道，但不改变 position，也就起到了扩充通道的作用
        need -= fillCapacity;
        size += fillCapacity;
        total += fillCapacity;
    }
}
```

4. EditLogInputStream

介绍完了 EditLogOutputStream 的实现，我们再来看输入流，EditLogInputStream 类抽象了从持久化存储上读 editlog 文件的相关接口。EditLogInputStream 的继承关系如图 3-24 所示，与 EditLogOutputStream 的类结构相同，不同的存储系统有与之对应的输入流子类。

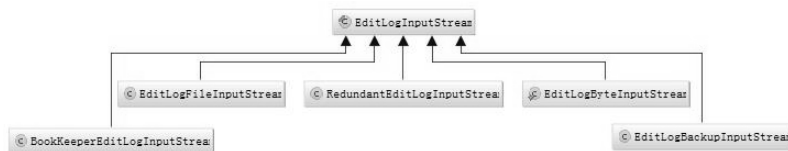


图 3-24 EditLogInputStream 继承关系图

这里以 EditLogFileInputStream 为例，EditLogFileInputStream 定义了本地文件系统的 editlog 文件的输入流。它定义的方法都很简单，都是返回了 EditLogFileInputStream 初始化以后的相应字段，或者调用了 FSEditLogOp.Reader 对象的 readOp()方法从 editlog 文件中解析出一个 FSEditLogOp 对象。这里就不详细分析了，大家可以自己阅读代码。

5. FSEditLog.log*()方法

介绍完 editlog 文件的输出流和输入流后，我们来看 FSEditLog 提供的 log*()方法。FSEditLog 类最重要的作用就是在 editlog 文件中记录 Namenode 命名空间的更改，FSEditLog 类对外提供了若干 log*()方法用于执行这个操作。如图 3-25 所示，log*()是 FSEditLog 中最多的方法，同时也是这个类的入口方法。

```

① logRename(String, String, long, boolean): void
② logRename(String, String, long, boolean, Rename...): void
③ logSetReplication(String, short): void
④ logSetStoragePolicy(String, byte): void
⑤ logSetQuota(String, long, long): void
⑥ logSetPermissions(String, FsPermission): void
⑦ logSetOwner(String, String, String): void
⑧ logConcat(String, String[], long, boolean): void
⑨ logDelete(String, long, boolean): void
⑩ logGenerationStampV1(long): void
⑪ logGenerationStampV2(long): void
⑫ logAllocateBlockId(long): void
⑬ logTimes(String, long, long): void
⑭ logSymlink(String, String, long, long, INodeSymlink, boolean): void
⑮ logGetDelegationToken(DelegationTokenIdentifier, long): void
⑯ logRenewDelegationToken(DelegationTokenIdentifier, long): void
⑰ logCancelDelegationToken(DelegationTokenIdentifier): void
⑱ logUpdateMasterKey(DelegationKey): void
⑲ logReassignLease(String, String, String): void
⑳ logCreateSnapshot(String, String, boolean): void
㉑ logDeleteSnapshot(String, String, boolean): void
㉒ logRenameSnapshot(String, String, String, boolean): void
㉓ logAllowSnapshot(String): void
㉔ logDisallowSnapshot(String): void
㉕ logAddCacheDirectiveInfo(CacheDirectiveInfo, boolean): void
㉖ logModifyCacheDirectiveInfo(CacheDirectiveInfo, boolean): void
㉗ logRemoveCacheDirectiveInfo(Long, boolean): void
㉘ logAddCachePool(CachePoolInfo, boolean): void
㉙ logModifyCachePool(CachePoolInfo, boolean): void
㉚ logRemoveCachePool(String, boolean): void
㉛ logStartRollingUpgrade(long): void

```

图 3-25 log*()方法

这里我们以 logDelete()方法为例，logDelete()方法用于在 editlog 文件中记录删除 HDFS 文件的操作。logDelete()方法首先会构造一个 DeleteOp 对象，这个 DeleteOp 类是 FSEditLogOp 类的子类，用于记录删除操作的相关信息，包括了 ClientProtocol.delete()调用中所有参数携带的信息。上面提到的 FSEditLogOp 类是一个抽象的工具类，它定义了 editlog 记录的操作类型，并且提供了从 editlog 输入流中解析操作参数等功能。每个 editlog 日志文件中可以记录的操作，都有一个与之对应的 FSEditLogOp 的子类用来记录这个操作的信息，例如 delete()操作对应 DeleteOp 类。成功构造 DeleteOp 对象后，logDelete()方法会调用 logRpcIds()方法在 DeleteOp 对象中添加 RPC 调用相关信息，之后 logDelete()方法会调用 logEdit()方法在 editlog 文件中记录这次删除操作。logDelete()方法的代码如下：

```

void logDelete(String src, long timestamp, boolean toLogRpcIds) {
    DeleteOp op = DeleteOp.getInstance(cache.get()) // 构造 DeleteOp 对象
    .setPath(src)
    .setTimestamp(timestamp);
}

```

Hadoop 2.X HDFS 源码剖析

```
logRpcIds(op, toLogRpcIds); // 记录 RPC 调用相关信息, 参考下面的 logRpcIds() 方法分析
logEdit(op); // 调用 logEdit() 方法记录删除操作
}

private void logRpcIds(FSEditLogOp op, boolean toLogRpcIds) {
    if (toLogRpcIds) {
        op.setRpcClientId(Server.getClientId());
        op.setRpcCallId(Server.getCallId());
    }
}
```

通过分析源码可知,基本上所有的 `log*()` 方法 (例如 `logDelete()`、`logCloseFile()`) 在底层都调用了 `logEdit()` 方法来执行记录操作,这里会传入一个 `FSEditLogOp` 对象来标识当前需要被记录的操作类型以及操作的信息。我们先看一下 `logEdit()` 方法的实现,`logEdit()` 方法会调用 `beginTransaction()` 方法在 `editlog` 文件中开启一个新的 `transaction`,然后使用 `editlog` 输入流写入要被记录的操作,接下来调用 `endTransaction()` 方法关闭这个 `transaction`,最后调用 `logSync()` 方法将写入的信息同步到磁盘上。`logEdit()` 方法的代码如下:

```
void logEdit(final FSEditLogOp op) {
    synchronized (this) {
        // 如果自动同步开启,则等待同步完成
        waitIfAutoSyncScheduled();
        // 开启一个新的 transaction
        long start = beginTransaction();
        op.setTransactionId(txid);
        // 使用 editLogStream 写入 Op 操作
        try {
            editLogStream.write(op);
        } catch (IOException ex) {
        }
        // 结束当前的 transaction
        endTransaction(start);
        // 检查是否需要强制同步
        if (!shouldForceSync()) {
            return;
        }
        isAutoSyncScheduled = true;
    }
    // 同步当前写入的操作,持久化到硬盘上
    logSync();
}
```

需要注意的是, `logEdit()` 方法调用 `beginTransaction()`、`editLogStream.write()` 以及 `endTransaction()` 三个方法时使用了 `synchronized` 关键字进行同步操作,这样就保证了多个线程调用 `FSEditLog.log*()` 方法向 `editlog` 文件中写数据时, `editlog` 文件记录的内容不会相互影响。同时,也保证了这几个并发线程保存操作对应的 `transactionId` (通过调用 `beginTransaction()` 方法获得,在后面小节中会介绍这个方法) 是唯一并递增的。

细心的读者可能会发现, `logEdit()`方法中调用 `logSync()`方法执行刷新操作的语句并不在 `synchronized` 代码段中。这是因为调用 `logSync()`方法必然会触发写 `editlog` 文件的磁盘操作, 这是一个非常耗时的操作, 如果放入同步模块中会造成其他调用 `FSEditLog.log*()`线程的等待时间过长。所以, HDFS 设计者将需要进行同步操作的 `synchronized` 代码段放入 `logSync()`方法中, 也就让输出日志记录和刷新缓冲区数据到磁盘这两个操作分离了。同时, 利用 `EditLogOutputStream` 的两个缓冲区, 使得日志记录和刷新缓冲区数据这两个操作可以并发执行, 大大地提高了 Namenode 的吞吐量。这是一个非常巧妙的设计, 值得读者朋友们积累。

下面我们就具体分析 `logEdit()`调用的几个方法, 其中 `editLogStream.write()`方法在输出流小节中已经介绍了 (请参考 `EditLogFileOutputStream` 小节), 下面我们分别介绍 `beginTransaction()`、`endTransaction()`、`logSync()`等方法的实现。

(1) `beginTransaction()`

`logEdit()`方法会调用 `beginTransaction()`方法开启一个新的 `transaction`, 也就是将 `FSEditLog.txid` 字段增加 1 并作为当前操作的 `transactionId`。`FSEditLog.txid` 字段维护了一个全局递增的 `transactionId`, 这样也就保证了 `FSEditLog` 为所有操作分配的 `transactionId` 是唯一且递增的。调用 `beginTransaction()`方法之后会将新申请的 `transactionId` 放入 `ThreadLocal` 的变量 `myTransactionId` 中, `myTransactionId` 保存了当前线程记录操作对应的 `transactionId`, 方便了以后线程做 `sync` 同步操作。`beginTransaction()`的代码如下:

```
private long beginTransaction() {
    assert Thread.holdsLock(this);
    // 全局的 transactionId ++
    txid++;

    // 使用 ThreadLocal 变量保存当前线程持有的 transactionId
    TransactionId id = myTransactionId.get();
    id.txid = txid;
    return now();
}
```

注意, 对于 `FSEditLog` 类, 可能同时有多个线程并发地调用 `log*()`方法执行日志记录操作, 所以 `FSEditLog` 类使用了一个 `ThreadLocal` 变量 `myTransactionId` 为每个调用 `log*()`操作的线程保存独立的 `txid`, 这个 `txid` 为当前线程记录操作对应的 `transactionId`。

注: `ThreadLocal` 变量会为每个线程提供一个独立的变量副本, 从而隔离了多个线程对数据的访问冲突。因为每个线程都拥有自己的变量副本, 从而也就没有必要对该变量进行同步了。

(2) `endTransaction()`

`logEdit()`方法会调用 `endTransaction()`方法结束一个 `transaction`, 这个方法的实现很简单, 就是更改一些统计数据, 这里就不做具体解释了。

(3) logSync()

logEdit()方法通过调用 beginTransaction()方法成功地获取一个 transactionId 之后，就会通过输出流向 editlog 文件写数据以记录当前的操作，但是写入的这些数据并没有直接保存在 editlog 文件中，而是暂存在输出流的缓冲区中。所以当 logEdit()方法将一个完整的操作写入输出流后，需要调用 logSync()方法同步当前线程对 editlog 文件所做的修改。我们知道会有多个线程同时写 editlog 文件，所以 editlog 制订了以下同步策略。

- 所有的操作项同步地写入缓存时，每个操作会被赋予一个唯一的 transactionId。
- 当一个线程要将它的操作同步到 editlog 文件中时，logSync()方法会使用 ThreadLocal 变量 myTransactionId 获取该线程需要同步的 transactionId，然后对比这个 transactionId 和已经同步到 editlog 文件中的 transactionId。如果当前线程的 transactionId 大于 editlog 文件中的 transactionId，则表明 editlog 文件中记录的数据不是最新的，同时如果当前没有别的线程执行同步操作，则开始同步操作将输出流缓存中的数据写入 editlog 文件中。需要注意的是，由于 editlog 输出流使用了双 buffer 的结构，所以在进行 sync 操作的同时，并不影响 editlog 输出流的使用。
- 在 logSync()方法中使用 isSyncRunning 变量标识当前是否有线程正在进行同步操作，这里注意 isSyncRunning 是一个 volatile 的 boolean 类型变量。

logSync()方法分为以下三个部分，并分开进行加锁操作，这样的设计提高了并发的程度。

- 判断当前操作是否已经同步到了 editlog 文件中，如果还没有同步，则将 editlog 的双 buffer 调换位置，为同步操作做准备，同时将 isSyncRunning 标志位设置为 true，这部分代码需要进行 synchronized 加锁操作。
- 调用 logStream.flush()方法将缓存的数据持久化到存储上，这部分代码不需要进行加锁操作，因为在上一段同步代码中已经将双 buffer 调换了位置，不会有线程向用于刷新数据的缓冲区中写入数据，所以调用 flush()操作并不需要加锁。
- 重置 isSyncRunning 标志位，并且通知等待的线程，这部分代码需要进行 synchronized 加锁操作。

logSync()方法的代码如下：

```
public void logSync() {
    long syncStart = 0;

    // ThreadLocal 保存的当前线程需要同步的 txid
    long mytxid = myTransactionId.get().txid;

    boolean sync = false;
    try {
        EditLogOutputStream logStream = null;

        // 第一部分，头部代码
        synchronized (this) {
            try {
```

```

printStatistics(false);

// 当前 txid 大于 editlog 中已经同步的 txid, 并且有线程正在同步, 则等待
while (mytxid > synctxid && isSyncRunning) {
    try {
        wait(1000);
    } catch (InterruptedException ie) {
    }
}

// 如果 txid 小于 editlog 中已经同步的 txid, 则表明当前操作已经被同步到存储上, 不需要再次同步
if (mytxid <= synctxid) {
    numTransactionsBatchedInSync++;
    //...
    return;
}

// 否则开始同步操作, 将 isSyncRunning 标志位设置为 true
syncStart = txid;
isSyncRunning = true;
sync = true;

//通过调用 setReadyToFlush() 方法将两个缓冲区互换, 为同步做准备
try {
    if (journalSet.isEmpty()) {
        throw new IOException("No journals available to flush");
    }
    editLogStream.setReadyToFlush();
} catch (IOException e) {
    // 异常处理 ...
}
} finally {
    doneWithAutoSyncScheduling();
}
logStream = editLogStream;
}

// 第二部分, 调用 flush() 方法, 将缓存中的数据同步到 editlog 文件中
long start = now();
try {
    if (logStream != null) {
        logStream.flush();
    }
} catch (IOException ex) {
    synchronized (this) {
        IOUtils.cleanup(LOG, journalSet);
        terminate(1, msg);
    }
}

```

```
    }
    long elapsed = now() - start;
} finally {
    // 第三部分，恢复标志位
    synchronized (this) {
        if (sync) {
            // 已同步 txid 赋值为开始 sync 操作的 txid
            syncTxid = syncStart;
            isSyncRunning = false;
        }
        this.notifyAll();
    }
}
```

了解了 `logSync()` 的加锁机制，我们再来看 `logEdit()` 和 `logSync()` 方法的同步机制。由于 `logEdit()` 方法中输出日志记录和调用 `logSync()` 刷新缓冲区数据到磁盘这两个操作是独立加锁的，同时 `EditLogOutputStream` 提供了两个缓冲区可以同时进行日志记录和刷新缓冲区操作，所以 `logEdit()` 方法中使用 `synchronized` 关键字同步的日志记录操作和 `logSync()` 方法中使用 `synchronized` 关键字同步的刷新缓冲区数据到磁盘的操作是可以并发同步进行的，它们都使用 `FSEditLog` 对象作为锁对象。这种设计大大地提高了多个线程记录 `editlog` 操作的并发性，且通过 `transactionId` 机制保证了 `editlog` 日志记录的正确性，是一个非常巧妙的优化。

3.1.4 FSImage 类

通过前面的介绍我们知道，`Namenode` 会定期将文件系统的命名空间（文件目录树、文件/目录元信息）保存到 `fsimage` 文件中，以防止 `Namenode` 掉电或者进程崩溃。但如果 `Namenode` 实时地将内存中的元数据同步到 `fsimage` 文件中，将会非常消耗资源且造成 `Namenode` 运行缓慢。所以 `Namenode` 会先将命名空间的修改操作保存在 `editlog` 文件中，然后定期合并 `fsimage` 和 `editlog` 文件。

在上一节中我们介绍了负责 `editlog` 文件操作的 `FSEditLog` 类，本节介绍管理 `fsimage` 文件的实现类 `FSImage`，`FSImage` 类主要实现了以下功能。

- 保存命名空间——将当前时刻 `Namenode` 内存中的命名空间保存到 `fsimage` 文件中。
- 加载 `fsimage` 文件——将磁盘上 `fsimage` 文件中保存的命名空间加载到 `Namenode` 内存中，这个操作是保存命名空间操作的逆操作。
- 加载 `editlog` 文件——`Namenode` 加载了 `fsimage` 文件后，内存中只包含了命名空间在保存 `fsimage` 文件时的信息，`Namenode` 还需要加载后续对命名空间的修改操作，即 `editlog` 文件中记录的内容。所以 `FSImage` 类还提供了加载 `editlog` 文件到 `Namenode` 内存中的功能。

下面我们依次学习 `FSImage` 中这三个功能的具体实现。

1. 保存命名空间

FSImage 类最重要的功能之一就是当前时刻 Namenode 的命名空间保存到 fsimage 文件中。本节就介绍 FSImage 类中这个功能的实现，保存命名空间操作的调用顺序如图 3-26 所示。

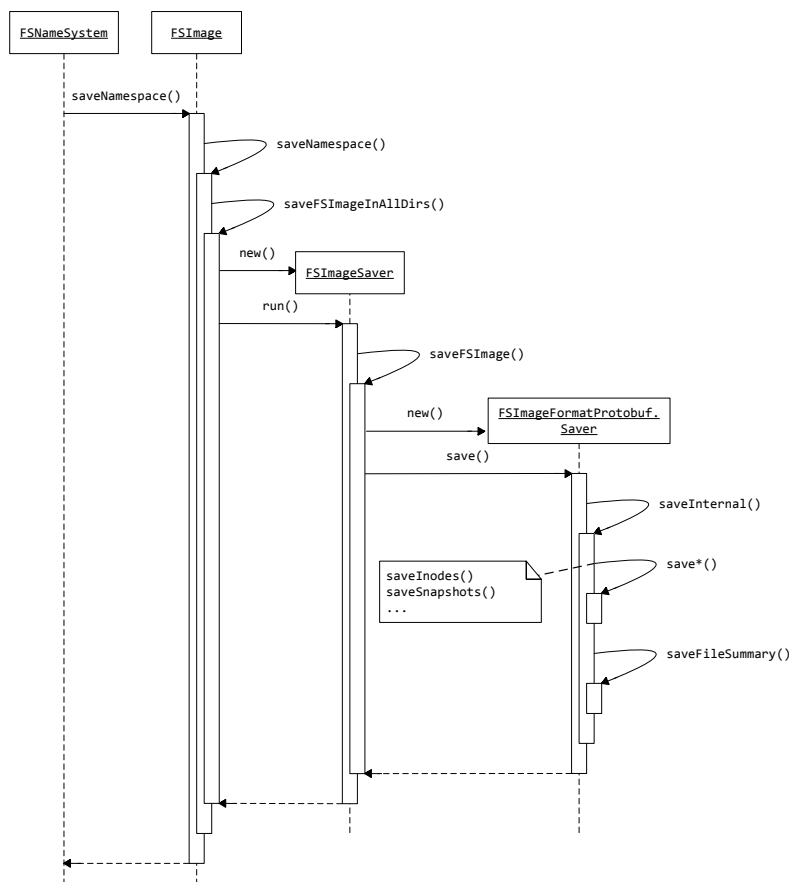


图 3-26 saveNamespace()调用流程

FSNameSystem 会调用 FSImage.saveNamespace() 方法触发命名空间的保存操作，saveNamespace()会调用 saveFSImageInAllDirs()方法执行具体的保存逻辑。本节我们从入口方法 saveFSImageInAllDirs()开始介绍。

(1) saveFSImageInAllDirs()

Namenode 可以定义多个存储路径来保存 fsimage 文件，对于每一个存储路径，saveFSImageInAllDirs()方法都会启动一个线程负责在这个路径上保存 fsimage 文件。同时，为

Hadoop 2.X HDFS 源码剖析

为了防止保存过程中出现错误，命名空间信息首先会被保存在一个 `fsimage.ckpt` 文件中，当保存操作全部完成之后，才会将 `fsimage.ckpt` 重命名为 `fsimage` 文件。之后 `saveFSImageInAllDirs()` 方法会清理 `Namenode` 元数据存储文件夹中过期的 `editlog` 文件和 `fsimage` 文件。`saveFSImageInAllDirs()` 方法的代码如下：

```
private synchronized void saveFSImageInAllDirs(FSNamesystem source,
        NameNodeFile nnf, long txid, Canceler canceler) throws IOException {
    // ...
    // 构造保存命名空间操作的上下文
    SaveNamespaceContext ctx = new SaveNamespaceContext(
        source, txid, canceler);

    try {
        List<Thread> saveThreads = new ArrayList<Thread>();
        // 在每一个保存路径上启动一个线程，该线程使用 FSImageSaver 类保存 fsimage 文件
        for (Iterator<StorageDirectory> it
            = storage.dirIterator(NameNodeDirType.IMAGE); it.hasNext();) {
            StorageDirectory sd = it.next();
            FSImageSaver saver = new FSImageSaver(ctx, sd, nnf);
            Thread saveThread = new Thread(saver, saver.toString());
            saveThreads.add(saveThread);
            saveThread.start();
        }
        // 等待所有线程执行完毕
        waitForThreads(saveThreads);
        saveThreads.clear();
        storage.reportErrorsOnDirectories(ctx.getErrorSDs());
        // 保存文件失败则抛出异常
        if (storage.getNumStorageDirs(NameNodeDirType.IMAGE) == 0) {
            throw new IOException(
                "Failed to save in any storage directories while saving namespace.");
        }
        // 将 fsimage.ckpt 改名为 fsimage
        renameCheckpoint(txid, NameNodeFile.IMAGE_NEW, nnf, false);

        // 我们已经完成了 fsimage 的保存，那么可以将存储上的一部分 editlog 和 fsimage 删除
        purgeOldStorage(nnf);
    } finally {
        // 通知所有等待的线程
        ctx.markComplete();
        ctx = null;
    }
    prog.endPhase(Phase.SAVING_CHECKPOINT);
}
```

通过分析 `saveFSImageInAllDirs()` 方法可知，命名空间具体的保存操作是由 `FSImageSaver` 这个类来承担的，`FSImageSaver` 是 `FSImage` 中的内部类，也是一个线程类，它的 `run()` 方法调用了 `saveFSImage()` 方法来保存 `fsimage` 文件。我们再看一下 `saveFSImage()` 方法的实现。

saveFSImage()方法会使用一个 FSImageFormat.Saver 对象来完成保存操作, FSImageFormat.Saver 类会以 fsimage 文件定义的格式保存 Namenode 的命名空间信息, 需要注意命名空间信息会先写入 fsimage.ckpt 文件中。saveFSImage()方法还会生成 fsimage 文件的 md5 校验文件, 以确保 fsimage 文件的正确性。saveFSImage()方法的代码如下:

```
void saveFSImage(SaveNamespaceContext context, StorageDirectory sd)
    throws IOException {
    long txid = context.getTxId(); // 获取当前命名空间中记录的最新事务的 txid
    // fsimage 文件
    File newFile = NNStorage.getStorageFile(sd, NameNodeFile.IMAGE_NEW, txid);
    File dstFile = NNStorage.getStorageFile(sd, NameNodeFile.IMAGE, txid);
    // FSImageFormatProtobuf.Saver 类负责保存 fsimage
    FSImageFormatProtobuf.Saver saver = new FSImageFormatProtobuf.Saver(context);
    FSImageCompression compression = FSImageCompression.createCompression(conf); // 压缩类
    saver.save(newFile, compression); // 调用 Saver 类保存 fsimage 文件

    MD5FileUtils.saveMD5File(dstFile, saver.getSavedDigest()); // 保存 MD5 校验值
    storage.setMostRecentCheckpointInfo(txid, Time.now());
}
```

saveFSImage()方法构造了一个 FSImageFormatProtobuf.Saver 对象来保存命名空间, FSImageFormatProtobuf 是一个工具类, 它提供了以 protobuf 格式读取和写入 fsimage 文件的方法。在 HDFS2.4 版本的实现中, FSImage 使用 FSImageFormat 类作为读取和写入 fsimage 文件的标准格式类, 而 HDFS2.6 版本则使用了 FSImageFormatProtobuf 替代了 FSImageFormat 类。FSImageFormatProtobuf 除了有 Saver 内部类用于保存命名空间到 fsimage 文件外, 还提供了一个 Loader 内部类用于解析和加载 fsimage 文件。这里我们先学习 FSImageFormatProtobuf.Saver 类的实现。

(2) FSImageFormatProtobuf.Saver

HDFS 2.6 版本使用了 protobuf 作为 fsimage 文件序列化的工具, fsimage 文件的格式也被重新定义了。图 3-27 给出了使用 protobuf 定义的 fsimage 文件的格式, 它包括了 4 个部分的信息。

- **MAGIC:** fsimage 的文件头, 是“HDFSIMG1”这个字符串的二进制形式, MAGIC 头标识了当前 fsimage 文件是使用 protobuf 格式序列化的。FSImage 类在读取 fsimage 文件时, 会先判断 fsimage 文件是否包含了 MAGIC 头, 如果包含了则使用 protobuf 格式反序列化 fsimage 文件。
- **SECTIONS:** fsimage 文件会将同一类型的 Namenode 元信息保存在一个 section 中, 例如将文件系统元信息保存在 NameSystemSection 中, 将文件系统目录树中的所有 INode 信息保存在 INodeSection 中, 将快照信息保存在 SnapshotSection 中等。fsimage 文件的第二个部分就是 Namenode 各类元信息对应的所有 section, fsimage 定义的 section 类型如图 3-27 所示, 每类 section 中都包含了对应 Namenode 元信息的属性, 每个 section 的格式定义请参考 fsimage.proto 文件。
- **FileSummary:** FileSummary 记录了 fsimage 文件的元信息, 以及 fsimage 文件保存的

所有 section 的信息。FileSummary 中的 ondiskVersion 字段记录了 fsimage 文件的版本号（2.6 版本中这个字段的值为 1），layoutVersion 字段记录了当前 HDFS 的文件系统布局版本号，codec 字段记录了 fsimage 文件的压缩编码，sections 字段则记录了 fsimage 文件中各个 section 字段的元信息，每个 fsimage 文件中记录的 section 在 FileSummary 中都有一个与之对应的 section 字段。FileSummary 的 section 字段记录了对应的 fsimage 中 section 的名称、在 fsimage 文件中的长度，以及这个 section 在 fsimage 中的起始位置。FSImage 类在读取 fsimage 文件时，会先从 fsimage 中读取 FileSummary 部分，然后利用 FileSummary 记录的元信息指导 fsimage 文件的反序列化操作。

- FileSummaryLength: FileSummaryLength 记录了 FileSummary 在 fsimage 文件中所占的长度，FSImage 类在读取 fsimage 文件时，会首先读取 FileSummaryLength 获取 FileSummary 部分的长度，然后根据这个长度从 fsimage 中反序列列出 FileSummary。

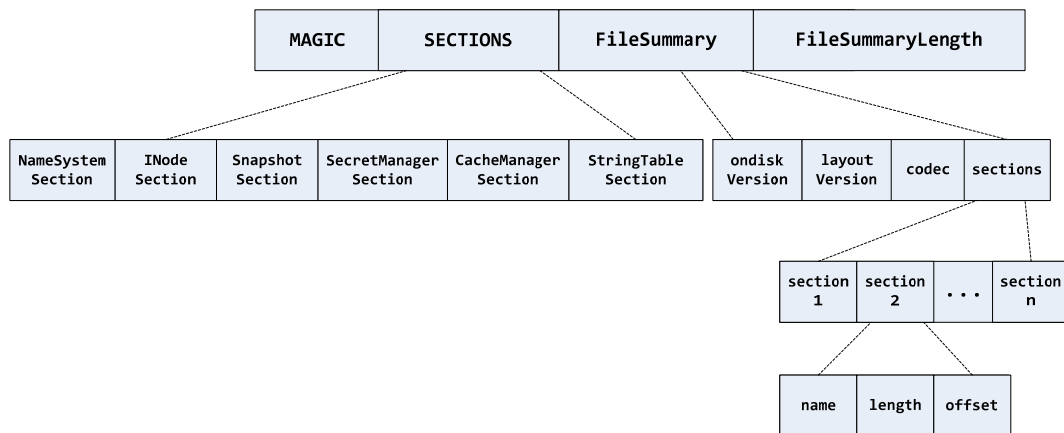


图 3-27 fsimage 文件格式

FSImageFormatProtobuf.Saver 类就是以 protobuf 格式将 Namenode 的命名空间保存至 fsimage 文件的工具类，如上一小节所述，这个类的入口方法是 save() 方法。save() 方法会打开 fsimage 文件的输出流并且获得文件通道，然后调用 saveInternal() 方法将命名空间保存到 fsimage 文件中。

saveInternal() 方法首先构造底层 fsimage 文件的输出流，构造 fsimage 文件的描述类 FileSummary，然后在 FileSummary 中记录 ondiskVersion、layoutVersion、codec 等信息。接下来 saveInternal() 方法依次向 fsimage 文件中写入命名空间信息、inode 信息、快照信息、安全信息、缓存信息、StringTable 信息等。注意上述信息都是以 section 为单位写入的，每个 section 的格式定义请参考 fsimage.proto 文件。saveInternal() 方法以 section 为单位写入元数据信息时，还会在 FileSummary 中记录这个 section 的长度，以及 section 在 fsimage 文件中的起始位置等信息。当完成了所有 section 的写入后，FileSummary 对象也就构造完毕了，saveInternal() 最后会将 FileSummary 对象写入 fsimage 文件中。saveInternal() 方法的代码如下：

```

private void saveInternal(FileOutputStream fout,
    FSImageCompression compression, String filePath) throws IOException {

    // 构造输出流, 一边写入数据, 一边写入校验值
    MessageDigest digester = MD5Hash.getDigester();
    underlyingOutputStream = new DigestOutputStream(new BufferedOutputStream(
        fout), digester);
    underlyingOutputStream.write(FSImageUtil.MAGIC_HEADER);
    fileChannel = fout.getChannel();

    // FileSummary 为 fsimage 文件的描述部分, 也是 protobuf 定义的
    FileSummary.Builder b = FileSummary.newBuilder()
        .setOnDiskVersion(FSImageUtil.FILE_VERSION)
        .setLayoutVersion(NameNodeLayoutVersion.CURRENT_LAYOUT_VERSION);
    codec = compression.getImageCodec(); // 获取压缩格式, 并装饰输出流
    if (codec != null) {
        b.setCodec(codec.getClass().getCanonicalName());
        sectionOutputStream = codec.createOutputStream(underlyingOutputStream);
    } else {
        sectionOutputStream = underlyingOutputStream;
    }

    saveNameSystemSection(b); // 保存命名空间信息
    context.checkCancelled(); // 检查是否取消了保存操作

    saveINodes(b);           // 保存命名空间中的 inode 信息
    saveSnapshots(b);        // 保存快照信息
    saveSecretManagerSection(b); // 保存安全信息
    saveCacheManagerSection(b); // 保存缓存信息
    saveStringTableSection(b); // 保存 StringTable

    flushSectionOutputStream(); // flush 输出流

    FileSummary summary = b.build();
    saveFileSummary(underlyingOutputStream, summary); // 将 FileSummary 写入文件
    underlyingOutputStream.close(); // 关闭底层输出流
    savedDigest = new MD5Hash(digester.digest());
}

```

可以看到, `saveInternal()`方法调用了多个 `save*()`方法来记录不同 section 的元数据信息, 这些方法除了在 `fsimage` 文件中写入对应种类的元数据信息外, 还会在 `FileSummary` 中记录 section 的大小, 以及在 `fsimage` 中的起始位置。这里我们重点介绍 `saveINodes()`方法的实现。`saveINodes()`方法构造了一个 `FSImageFormatPBINode.Saver` 对象, 并调用这个对象对应的方法保存文件系统目录树中的 `INode` 信息、`INodeDirectory` 信息, 以及处于构建状态的文件信息。

```

private void saveINodes(FileSummary.Builder summary) throws IOException {
    FSImageFormatPBINode.Saver saver = new FSImageFormatPBINode.Saver(this,
        summary);
}

```

```
saver.serializeINodeSection(sectionOutputStream);
saver.serializeINodeDirectorySection(sectionOutputStream);
saver.serializeFilesUCSection(sectionOutputStream);
}
```

保存 INode 信息是由 FSImageFormatPBINode.Saver.serializeINodeSection()方法实现的。serializeINodeSection()方法会首先构造一个 INodeSection 对象，记录文件系统目录树中保存的最后一个 inode 的 inodeid，以及命名空间中所有 inode 的个数。之后迭代处理将所有 inode 信息写入 fsimage 文件中，最后将 INodeSection 的属性信息记录在 FileSummary 中。serializeINodeSection()方法的代码如下：

```
void serializeINodeSection(OutputStream out) throws IOException {
    INodeMap inodesMap = fsn.dir.getINodeMap();
    // 构造一个 INodeSection，保存最后一个 inode 的 inodeid，以及这个命名空间中所有 inode 的个数
    INodeSection.Builder b = INodeSection.newBuilder()
        .setLastINodeId(fsn.getLastINodeId()).setNumINodes(inodesMap.size());
    INodeSection s = b.build();
    // 序列化至输出流
    s.writeDelimitedTo(out);

    // 迭代处理 inodesMap 中所有的 inode，调用 save() 方法将 inode 信息保存到 fsimage 中
    int i = 0;
    Iterator<INodeWithAdditionalFields> iter = inodesMap.getMapIterator();
    while (iter.hasNext()) {
        INodeWithAdditionalFields n = iter.next();
        save(out, n);
        ++i;
        if (i % FSImageFormatProtobuf.Saver.CHECK_CANCEL_INTERVAL == 0) {
            context.checkCancelled();
        }
    }
    // 调用 commitSection() 方法在 FileSummary 中写入 inode section
    parent.commitSection(summary, FSImageFormatProtobuf.SectionName.INODE);
}
```

下面我们看一下 save()方法的实现。save()方法首先将当前 INode 对象分为目录、文件以及符号链接等几类，然后调用各个类型对应的 save()重载方法。重载方法的实现也非常简单，就是构造不同的 protobuf Builder 类，然后设置相应字段的值，并将序列化之后的对象写入 fsimage 文件的输出流中。这里以 INodeFile 为例，首先构造 protobuf Builder 类——INodeSection.INodeFile.Builder，然后设置 blocks——也就是当前文件有哪些数据块，如果当前的 INodeFile 处于构建状态，则设置对应的构建信息。最后将序列化后的 inode 信息写入输出流中。文件夹以及符号链接的写法在这里就不再描述了，感兴趣的读者可以自行阅读代码。save()方法的代码如下：

```
private void save(OutputStream out, INode n) throws IOException {
    if (n.isDirectory()) {
```

```

        save(out, n.asDirectory());
    } else if (n.isFile()) {
        save(out, n.asFile());
    } else if (n.isSymlink()) {
        save(out, n.asSymlink());
    }
}

private void save(OutputStream out, INodeFile n) throws IOException {
    INodeSection.INodeFile.Builder b = buildINodeFile(n,
        parent.getSaverContext());

    if (n.getBlocks() != null) {
        for (Block block : n.getBlocks()) {
            b.addBlocks(PBHelper.convert(block));
        }
    }

    FileUnderConstructionFeature uc = n.getFileUnderConstructionFeature();
    if (uc != null) {
        INodeSection.FileUnderConstructionFeature f =
            INodeSection.FileUnderConstructionFeature
                .newBuilder().setClientName(uc.getClientName())
                .setClientMachine(uc.getClientMachine()).build();
        b.setFileUC(f);
    }

    INodeSection.INode r = buildINodeCommon(n)
        .setType(INodeSection.INode.Type.FILE).setFile(b).build();
    r.writeDelimitedTo(out);
}

```

通过上面的分析，可以看到 HDFS2.6 版本使用 **protobuf** 替代了原有的 **fsimage** 文件的序列化方法，使得 **fsimage** 文件的保存代码变得更简洁、更可读。对于 **protobuf** 的使用，请读者注意积累，它可以作为元数据文件序列化的一种非常好的工具。

2. FSImage.loadFSImage()

当 Namenode 启动时，首先会将 **fsimage** 文件中记录的命名空间加载到 Namenode 内存中，然后再一条一条地将 **editlog** 文件中记录的更新操作加载并合并到命名空间中。接下来 Namenode 会等待各个 Datanode 向自己汇报数据块信息来组装 **blockMap**，从而离开安全模式。Namenode 每次启动时都会调用 **FSImage.loadFSImage()** 方法执行加载 **fsimage** 和 **editlog** 文件的操作。我们看一下 **loadFSImage()** 方法的代码。

```

boolean loadFSImage(FSNamesystem target, MetaRecoveryContext recovery)
    throws IOException {

    // 准备工作 ...

```

Hadoop 2.X HDFS 源码剖析

```
// 获取 editlog 文件 IO 流
initEditLog();
if (LayoutVersion.supports(Feature.TXID_BASED_LAYOUT,
                           getLayoutVersion())) {
    long toAtLeastTxId = editLog.isOpenForWrite() ? inspector.getMaxSeenTxId() : 0;
    editStreams = editLog.selectInputStreams(
        imageFiles.get(0).getCheckpointTxId() + 1,
        toAtLeastTxId, recovery, false);
} else {
    editStreams = FSImagePreTransactionalStorageInspector
        .getEditLogStreams(storage);
}
// ...

// 调用 loadFSImageFile() 方法, 加载 fsimage 文件
for (int i = 0; i < imageFiles.size(); i++) {
    try {
        imageFile = imageFiles.get(i);
        loadFSImageFile(target, recovery, imageFile);
        break;
    } catch (IOException ioe) {
        LOG.error("Failed to load image from " + imageFile, ioe);
        target.clear();
        imageFile = null;
    }
}
// 如果加载失败, 则抛出异常
if (imageFile == null) {
    FSEditLog.closeAllStreams(editStreams);
    throw new IOException("Failed to load an FSImage file!");
}

// 调用 loadEdit() 方法加载并合并 editlog
long txnsAdvanced = loadEdits(editStreams, target, recovery);
needToSave |= needsResaveBasedOnStaleCheckpoint(imageFile.getFile(),
                                                  txnsAdvanced);
editLog.setNextTxId(lastAppliedTxId + 1);
return needToSave;
}
```

在 `loadFSImage()` 方法中加载 `fsimage` 文件是通过调用 `FSImage.loadFSImageFile()` 方法实现的, 加载 `editlog` 文件则是通过调用 `FSImage.loadEdits()` 方法实现的。下面我们依次介绍在 `FSImage` 中加载 `fsimage` 和 `editlog` 文件的具体实现。

3. 加载 fsimage 文件

上一节我们介绍了 `FSImage.loadFSImage()` 方法会调用 `FSImage.loadFSImageFile()` 方法执行加载 `fsimage` 文件的操作。`loadFSImageFile()` 方法的调用顺序如图 3-28 所示。

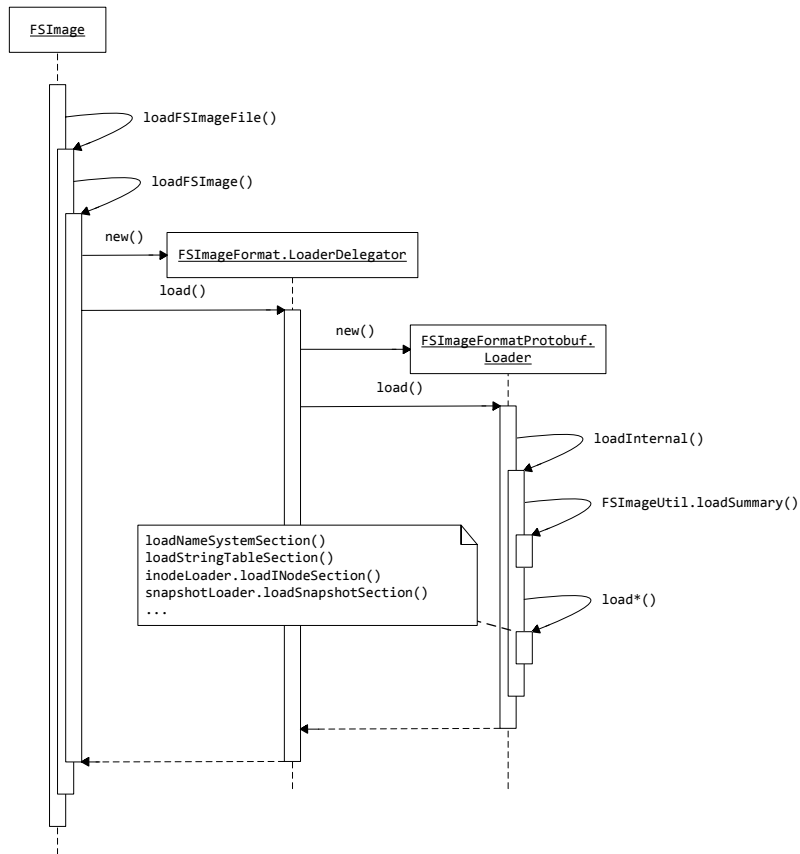


图 3-28 loadFSImageFile()方法的调用顺序

loadFSImageFile()方法会根据当前 Namenode 的版本调用不同的加载方法，不同版本加载方法的区别主要是在 fsimage 文件的校验操作上，例如对于 2.6 版本的加载操作，每个 fsimage 文件都有一个对应的 md5 校验文件。fsimage 文件最终的加载工作是由私有的 loadFSImage() 方法实现的，loadFSImage() 方法会构造一个 FSImageFormat.LoaderDelegator 对象加载 fsimage 文件，这个对象的 load() 方法会判断当前 fsimage 使用了什么序列化方式，如果在 fsimage 文件中有 MAGIC_HEADER，则该 fsimage 文件使用的是 protobuf 序列化方式，那么构造 FSImageFormatProtobuf.Loader 执行加载操作，否则使用 FSImageFormat.Loader 类执行加载操作。

load()的代码如下：

```
public void load(File file, boolean requireSameLayoutVersion)
    throws IOException {
    FileInputStream is = null;
```

```
try {
    is = new FileInputStream(file);
    byte[] magic = new byte[FSImageUtil.MAGIC_HEADER.length];
    IOUtils.readFully(is, magic, 0, magic.length);
    // fsimage 文件中包括 magicHeader, 使用的是 protobuf 序列化方式
    if (Arrays.equals(magic, FSImageUtil.MAGIC_HEADER)) {
        // 构造 FSImageFormatProtobuf.Loader 加载 fsimage 文件
        FSImageFormatProtobuf.Loader loader = new FSImageFormatProtobuf.Loader(
            conf, fsn, requireSameLayoutVersion);
        impl = loader;
        loader.load(file);
    } else {
        // 否则构造 FSImageFormat.Loader 加载 fsimage 文件
        Loader loader = new Loader(conf, fsn);
        impl = loader;
        loader.load(file);
    }
} finally {
    IOUtils.cleanup(LOG, is);
}
}
```

在保存命名空间小节中我们已经介绍过了, HDFS2.6 版本使用 protobuf 作为 fsimage 的序列化工具。所以在加载 fsimage 操作中, 最终会调用 FSImageFormatProtobuf.Loader 作为 fsimage 文件的加载类。FSImageFormatProtobuf.Loader.loadInternal() 方法执行了加载 fsimage 文件的操作, loadInternal() 方法会打开 fsimage 文件通道, 然后读取 fsimage 文件中的 FileSummary 对象, FileSummary 对象中记录了 fsimage 中保存的所有 section 的信息。loadInternal() 会对 FileSummary 对象中保存的 section 排序, 然后遍历每个 section 并调用对应的方法从 fsimage 文件中加载这个 section。loadInternal() 方法的代码如下:

```
private void loadInternal(RandomAccessFile raFile, FileInputStream fin)
    throws IOException {
    // 从 fsimage 文件末尾加载 FileSummary, 也就是 fsimage 文件内容的描述
    FileSummary summary = FSImageUtil.loadSummary(raFile);
    // 获取通道
    FileChannel channel = fin.getChannel();

    // 构造 FSImageFormatPBINode.Loader 和 FSImageFormatPBSnapshot.Loader 加载 INode 以及
    Snapshot
    FSImageFormatPBINode.Loader inodeLoader = new FSImageFormatPBINode.Loader(
        fsn, this);
    FSImageFormatPBSnapshot.Loader snapshotLoader = new FSImageFormatPBSnapshot.Loader(
        fsn, this);

    // 对 fsimage 文件描述中记录的 sections 进行排序
    ArrayList<FileSummary.Section> sections = Lists.newArrayList(summary
        .getSectionsList());
}
```

```

Collections.sort(sections, new Comparator<FileSummary.Section>() {
    @Override
    public int compare(FileSummary.Section s1, FileSummary.Section s2) {
        SectionName n1 = SectionName.fromString(s1.getName());
        SectionName n2 = SectionName.fromString(s2.getName());
        if (n1 == null) {
            return n2 == null ? 0 : -1;
        } else if (n2 == null) {
            return -1;
        } else {
            return n1.ordinal() - n2.ordinal();
        }
    }
});

// 遍历每个 section, 并调用对应的方法加载这个 section
for (FileSummary.Section s : sections) {
    channel.position(s.getOffset()); // 在通道中定位这个 section 的起始位置
    InputStream in = new BufferedInputStream(new LimitInputStream(fin,
        s.getLength()));

    in = FSImageUtil.wrapInputStreamForCompression(conf,
        summary.getCodec(), in);

    String n = s.getName();

    switch (SectionName.fromString(n)) { // 调用对应的方法加载不同的 section
    case NS_INFO:
        loadNameSystemSection(in);
        break;
    case STRING_TABLE:
        loadStringTableSection(in);
        break;
    case INODE: {
        currentStep = new Step(StepType.INODES);
        inodeLoader.loadINodeSection(in);
    }
        break;
    case INODE_REFERENCE:
        snapshotLoader.loadINodeReferenceSection(in);
        break;
    case INODE_DIR:
        inodeLoader.loadINodeDirectorySection(in);
        break;
    case FILES_UNDERCONSTRUCTION:
        inodeLoader.loadFilesUnderConstructionSection(in);
        break;
    case SNAPSHOT:

```

```
        snapshotLoader.loadSnapshotSection(in);
        break;
    case SNAPSHOT_DIFF:
        snapshotLoader.loadSnapshotDiffSection(in);
        break;
    case SECRET_MANAGER: {
        loadSecretManagerSection(in);
    }
    break;
    case CACHE_MANAGER: {
        loadCacheManagerSection(in);
    }
    break;
    default:
        LOG.warn("Unrecognized section " + n);
        break;
    }
}
```

对于不同类型的 section, loadInternal()方法会调用不同的方法加载这个 section, 例如对于 InodeSection 会调用 InodeLoader.loadInodeSection()方法加载。load*()方法的实现都比较简单, 就是按照 protobuf 格式加载不同的 section, 由于篇幅原因这里就不再介绍这些方法了, 请读者自行参考源代码。

4. 加载 editlog 文件

Namenode 将 fsimage 中记录的特定时刻的命名空间镜像加载到内存之后, 还需要加载后续对命名空间的修改, 也就是 editlog 中记录的修改命名空间的操作。如 FSImage.loadFSImage() 小节所介绍的, Namenode 会调用 FSImage.loadEdits()方法将 editlog 文件中记录的更新操作与当前 Namenode 的命名空间进行合并。FSImage.loadEdits()方法会构造一个 FSEditLogLoader 对象, 然后遍历 Namenode 所有存储路径上保存的 editlog 文件的输入流 (EditLogInputStream, 请参考 FSEditLog 类小节), 并调用 FSEditLogLoader.loadFSEdits()方法加载指定路径上的 editlog 文件。loadEdits()方法的代码如下:

```
private long loadEdits(Iterable<EditLogInputStream> editStreams,
    FSNamesystem target, StartupOption startOpt, MetaRecoveryContext recovery)
    throws IOException {
    // 记录命名空间中加载的最新的事务 id
    long prevLastAppliedTxId = lastAppliedTxId;
    try {
        // 构造 FSEditLogLoader 对象用于加载 editlog 文件
        FSEditLogLoader loader = new FSEditLogLoader(target, lastAppliedTxId);

        // 遍历所有存储路径上 editlog 文件对应的输入流
        for (EditLogInputStream editIn : editStreams) {
            // 调用 FSEditLogLoader.loadFSEdits() 从某个存储路径上的 editlog 文件加载修改操作

```

```

    try {
        loader.loadFSEdits(editIn, lastAppliedTxId + 1, startOpt, recovery);
    } finally {
        // lastAppliedTxId 记录从 editlog 加载的最新的的事务 id
        lastAppliedTxId = loader.getLastAppliedTxId();
    }
    if (editIn.getLastTxId() != HdfsConstants.INVALID_TXID) {
        lastAppliedTxId = editIn.getLastTxId();
    }
}
} finally {
    // 关闭所有 editlog 文件的输入流
    FSEditLog.closeAllStreams(editStreams);
    updateCountForQuota(target.dir.rootDir);
}
return lastAppliedTxId - prevLastAppliedTxId;
}

```

FSEditLogLoader.loadFSEdits()会使用 EditLogInputStream 对象读取并加载 editlog 文件，这个的操作比较冗长，基本可以归纳为从 editLog 文件中读取一个操作，并使用 FSEditLogOp 对象封装，然后调用 applyEditLogOp()方法修改 Namenode 的命名空间。FSEditLogLoader.loadFSEdits()方法的简版代码如下：

```

// 从 editlog 文件中读取一个操作
FSEditLogOp op;
try {
    op = in.readOp();
    if (op == null) {
        break;
    }
} catch (Throwable e) {
    //...
}

//...

// 在当前命名空间中执行对应的修改操作
long inodeId = applyEditLogOp(op, fsDir, in.getVersion(), lastINodeId);
if (lastINodeId < inodeId) {
    lastINodeId = inodeId;
}

```

这里的 applyEditLogOp()方法就是对传入的 FSEditLogOp 对象做了一个 switch，然后在命名空间中执行 FSEditLogOp 对象记录的操作，非常简单，感兴趣的读者可以自己看源代码。

5. 检查点机制

一个正常大小的 editlog 文件往往在几十到几百个字节之间，但在某些极端的情况下，

editlog 文件会变得非常大,甚至将磁盘空间写满。通过上面小节的学习我们知道,在 Namenode 启动过程中一个很重要的部分就是逐条读取 editlog 文件中的记录,之后与 Namenode 命名空间合并。巨大的 editlog 文件会导致 Namenode 的启动时间过长,为了解决这个问题,HDFS 引入了检查点机制 (checkpointing)。

如图 3-29 所示,HDFS 的检查点机制会定时将 editlog 文件与 fsimage 文件合并以产生新的 fsimage 文件。这样 Namenode 就可以直接从 fsimage 加载元数据,而不用读取与合并 editlog 中的记录了。有了检查点机制,Namenode 命名空间的重建效率会大大提高,同时 Namenode 的启动时间也会相应减少。

但是合并 fsimage 与 editlog 以产生新的 fsimage 文件是一项非常消耗 I/O 和 CPU 资源的操作。在执行检查点操作期间,Namenode 还需要限制其他用户对 HDFS 的访问和操作。为了预防这种情况的发生,HDFS 将检查点操作从 Active Namenode 移动到了 Secondary Namenode 或者 Standby Namenode 上,至于具体是哪一种 Namenode,则取决于当前的 HDFS 是否开启了 HA 功能。

在以下两种情况下,Namenode 会触发一次检查点操作:① 超过了配置的检查点操作时长 (dfs.namenode.checkpoint.period 配置项配置);② 从上一次检查点操作后,发生的事务数 (transaction) 数超过了配置 (dfs.namenode.checkpoint.txns 配置项配置)。

下面我们学习 Secondary Namenode 和 Standby Namenode 实现的两种检查点机制。

(1) Secondary Namenode 执行检查点操作

在非 HA 部署环境下,检查点操作是由 Secondary Namenode 来执行的。Secondary Namenode 是 HDFS1.X 版本中一个非热备的 Namenode 备份节点,整个检查点流程如图 3-30 所示。

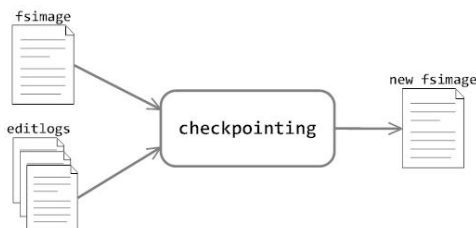


图 3-29 HDFS 检查点机制

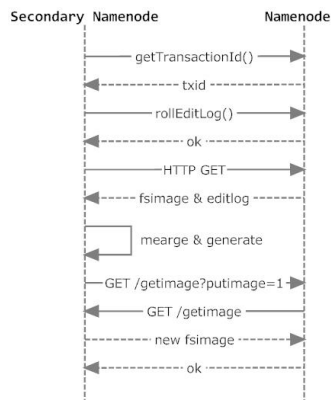


图 3-30 Secondary Namenode 检查点流程

整个检查点流程如下所示。

- Secondary Namenode 检查两个触发检查点流程的条件是否满足。由于在非 HA 状态

下, Secondary Namenode 和 Namenode 之间并没有共享的 editlog 文件目录, 所以最新的事务 id (transactionId) 是 Secondary Namenode 通过调用 RPC 方法 Namenode Protocol.getTransactionId() 获取的。

- Secondary Namenode 调用 RPC 方法 NamenodeProtocol.rollEditLog() 触发 editlog 重置操作, 将当前正在写的 editlog 段落结束, 并创建新的 edit.new 文件, 这个操作还会返回当前 fsimage 以及刚刚重置的 editlog 的事务 id (seen_id)。这样当 Secondary Namenode 从 Namenode 读取 editlog 文件时, 新的操作就可以写入 edit.new 文件中, 不影响 editlog 记录功能。在 HA 模式下, 并不需要显式地触发 editlog 的重置操作, 因为 Standby Namenode 会定期重置 editlog。
- 有了最新的 txid 以及 seen_id, Secondary Namenode 就会发起 HTTP GET 请求到 Namenode 的 GetImageServlet 以获取新的 fsimage 和 editlog 文件。需要注意, Secondary Namenode 在进行上一次的检查点操作时, 可能已经获取了部分 fsimage 和 edits 文件。
- Secondary Namenode 会加载新下载的 fsimage 文件以重建 Secondary Namenode 的命名空间。
- Secondary Namenode 读取 edits 中的记录, 并与当前的命名空间合并, 这样 Secondary Namenode 的命名空间和 Namenode 的命名空间就同步了。
- Secondary Namenode 将最新的同步的命名空间写入新的 fsimage 文件中。
- Secondary Namenode 向 Namenode 的 ImageServlet 发送 HTTP GET 请求/getimage?putimage=1。这个请求的 URL 中还包含了新的 fsimage 文件的事务 ID, 以及 Secondary Namenode 用于下载的端口和 IP 地址。
- Namenode 会根据 Secondary Namenode 提供的信息向 Secondary Namenode 的 GetImageServlet 发起 HTTP GET 请求下载 fsimage 文件。Namenode 首先将下载文件命名为 fsimage.ckpt_, 然后创建 MD5 校验和, 最后将 fsimage.ckpt_ 重命名为 fsimage_。

至此, 一个完整的 fsimage 的检查点操作就完成了。

(2) Standby Namenode 执行检查点操作

HDFS2.X 版本中加入了 Namenode HA 策略, 这使得 HA 机制下检查点操作的流程与非 HA 机制下的完全不同, 因为在 HA 配置下已经没有 Secondary Namenode 这个节点了, 而是直接通过配置奇数个 JournalNode 来实现 Namenode 热备 HA 策略。

这里我们首先介绍 HDFS2.X 版本中的 HA 策略, 图 3-31 给出了一个典型的 HA HDFS 集群结构图。在同一个 HA HDFS 集群中, 将会同时运行两个 Namenode 实例, 其中一个为 Active Namenode, 用于实时处理所有客户端请求; 另一个为 Standby Namenode, Standby Namenode 的命名空间与 Active Namenode 是完全保持一致的。所以当 Active Namenode 出现故障时, Standby Namenode 可以立即切换成 Active 状态。

为了让 Standby Namenode 的命名空间与 Active Namenode 保持同步, 它们都将和 JournalNodes 守护进程通信。当 Active Namenode 执行任何修改命名空间的操作时, 它至少需要将产生的 editlog 文件持久化到 $N-(N-1)/2$ 个 JournalNode 节点上才能保证命名空间修改的安全性。换句话说, 如果在 HA 策略下启动了 N 个 JournalNode 进程, 那么整个 JournalNode 集

群最多允许 $(N-1)/2$ 个进程死掉,这样才能保证 editlog 成功完整地写入。比如集群中有 3 个 JournalNode 时,最多允许 1 个 JournalNode 挂掉;集群中有 5 个 JournalNode 时,最多允许 2 个 JournalNode 挂掉。Standby Namenode 则负责观察 editlog 文件的变化,它能够从 JournalNodes 中读取 editlog 文件,然后合并更新到它自己的命名空间中。一旦 Active Namenode 出现故障,Standby Namenode 就会保证从 JournalNodes 中读出全部的 editlog 文件,然后切换到 Active 状态。Standby Namenode 读取全部的 editlog 文件可确保在发生故障转移之前,和 Active Namenode 拥有完全同步的命名空间状态。

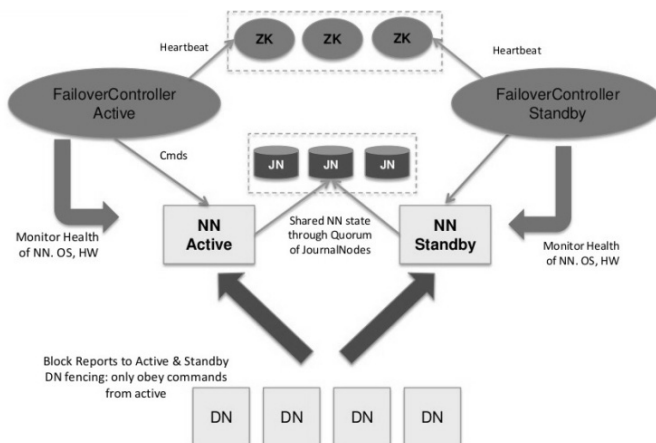


图 3-31 HA HDFS 集群结构图

Standby Namenode 始终保持着一个最新版本的命名空间,它会不断地将读入的 editlog 文件与当前的命名空间合并,所以检查点机制在 HA 模式下就简单了很多。Standby Namenode 只需判断当前是否满足触发检查点操作的两个条件,如果满足触发条件,则将 Standby Namenode 的命名空间写入一个新的 fsimage 文件中,并通过 HTTP 将这个 fsimage 文件传回 Active Namenode。HA 状态下的 Standby Namenode 检查点流程请参考图 3-32。

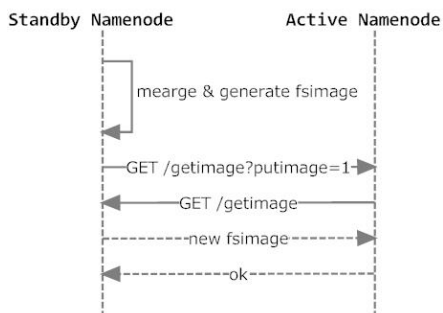


图 3-32 Standby Namenode 检查点流程

- Standby Namenode 检查是否满足触发检查点操作的两个条件。
- Standby Namenode 将当前的命名空间保存到 fsimage.ckpt_txid 文件中,这里的 txid

是当前最新的 editlog 文件中记录的事务 id。之后 Standby Namenode 写入 fsimage 文件的 MD5 校验和, 最后重命名这个 fsimage.ckpt_txid 文件为 fsimage_txid。当执行这个操作时, 其他的 Standby Namenode 操作将会被阻塞, 例如 Active Namenode 发生错误时, 需要进行主备切换或者访问 Standby Namenode 的 Web 接口等操作。注意, Active Namenode 的操作并不会被影响, 例如 listing、reading、writing 文件等。

- Standby Namenode 向 Active Namenode 的 ImageServlet 发送 HTTP GET 请求 /getimage?putimage=1。这个请求的 URL 中包含了新的 fsimage 文件的事务 id, 以及 Standby Namenode 用于下载的端口和 IP 地址。
- Active Namenode 会根据 Standby Namenode 提供的信息向 Standby Namenode 的 ImageServlet 发起 HTTP GET 请求下载 fsimage 文件。Namenode 首先将下载文件命名为 fsimage.ckpt_*, 然后创建 MD5 校验和, 最后将 fsimage.ckpt_*重命名为 fsimage_*。

知道了检查点操作的流程之后, 我们看一下 HDFS 代码是如何实现检查点功能的。StandbyNamenode 会持有一个 StandbyCheckpointeer 类, 这个类维护着一个叫作 CheckpointerThread 的线程, CheckpointerThread 线程会每隔 $1000 * \text{Math.min}(\text{checkpointCheckPeriod}, \text{checkpointPeriod})$ 秒检测是否执行一次检查点逻辑 (checkpointCheckPeriod 由 dfs.namenode.checkpoint.period 配置, checkpointPeriod 则由 dfs.namenode.checkpoint.check.period 配置)。整个检查点逻辑是在 CheckpointerThread.doWork()方法中实现的, 流程与上面介绍的一样。doWork()方法首先会判断是否满足检查点操作的两个条件, 如果满足则调用 doCheckpoint()执行检查点操作。

```
private void doWork() {
    while (shouldRun) {
        try {
            // 间隔时长 1000*Math.min(checkpointCheckPeriod, checkpointPeriod)
            Thread.sleep(1000 * checkpointConf.getCheckPeriod());
        } catch (InterruptedException ie) {
        }
        if (!shouldRun) {
            break;
        }
        try {
            long now = now();
            // 获得最后一次往 JournalNode 写入的 txid 和最近一次做检查点的 txid 的差值
            long uncheckpointed = countUncheckpointedTxns();
            // 计算当前时间和上一次检查点操作时间的间隔
            long secsSinceLast = (now - lastCheckpointTime)/1000;
            boolean needCheckpoint = false;
            // 第一种符合合并的情况: 当最后一次往 JournalNode 写入的 txid 和最近一次做检查点的 txid 的
            // 差值大于或者等于 dfs.namenode.checkpoint.txns 配置的数量 (默认为 1000000) 时做一次合并
            if (uncheckpointed >= checkpointConf.getTxnCount()) {
                needCheckpoint = true;
            }
        }
    }
}
```

Hadoop 2.X HDFS 源码剖析

```
        // 第二种符合合并的情况：当时间间隔大于或者等于 dfs.namenode.checkpoint.period 配置的
        时间时做合并
        else if (secsSinceLast >= checkpointConf.getPeriod()) {
            needCheckpoint = true;
        }

        // 满足检查点执行条件，则调用 doCheckpoint() 方法执行检查点操作
        if (needCheckpoint) {
            doCheckpoint();
            lastCheckpointTime = now;
        }
    } catch (SaveNamespaceCancelledException ce) {
        canceledCount++;
    } catch (InterruptedException ie) {
        continue;
    } catch (Throwable t) {
    } finally {
        synchronized (cancelLock) {
            canceler = null;
        }
    }
}
}
```

可以看到，整个检查点执行操作的逻辑都是在 `doCheckpoint()` 方法中实现的。`doCheckpoint()` 方法首先获取当前保存的 `fsimage` 的 `prevCheckpointTxId`，然后获取最近更新的 `editlog` 的 `thisCheckpointTxId`，只有新的 `thisCheckpointTxId` 大于 `prevCheckpointTxId`，也就是当前命名空间有更新，但是并没有保存到新的 `fsimage` 文件中时，才有必要进行一次检查点操作。判断完成后，`doCheckpoint()` 会调用 `saveNamespace()` 方法将最新的命名空间保存到 `fsimage` 文件中。之后构造一个线程，将新产生的 `fsimage` 文件通过 HTTP 方式上传到 `ActiveNamenode` 中。

```
private void doCheckpoint() throws InterruptedException, IOException {
    // ...
    namesystem.longReadLockInterruptibly();
    try {
        // 获取当前 Standby Namenode 上保存的最新的 fsimage 对象
        FSImage img = namesystem.getFSImage();
        // 获取 fsimage 中保存的 txid，也就是上一次进行检查点操作时保存的 txid
        long prevCheckpointTxId = img.getStorage().getMostRecentCheckpointTxId();
        // 获取当前命名空间的最新的 txid，也就是收到的 editlog 的最新的 txid
        long thisCheckpointTxId = img.getLastAppliedOrWrittenTxId();

        // thisCheckpointTxId 一定大于 prevCheckpointTxId
        assert thisCheckpointTxId >= prevCheckpointTxId;
        // 如果相等则没有必要执行检查点操作，当前 fsimage 已经是最新的了
        if (thisCheckpointTxId == prevCheckpointTxId) {
            return;
        }
    }
}
```

```

    }

    if (namesystem.isRollingUpgrade()
        && !namesystem.getFSImage().hasRollbackFSImage()) {
        // 如果当前 Namenode 正在执行升级操作, 则创建 fsimage_rollback 文件
        imageType = NameNodeFile.IMAGE_ROLLBACK;
    } else {
        // 在正常情况下创建 fsimage 文件
        imageType = NameNodeFile.IMAGE;
    }

    // 调用 saveNamespace() 将当前命名空间保存到新的文件中
    img.saveNamespace(namesystem, imageType, canceler);

} finally {
    namesystem.longReadUnlock();
}

// 构造一个线程, 通过 HTTP 将 fsimage 上传到 Active Namenode 中
ExecutorService executor =
    Executors.newSingleThreadExecutor(uploadThreadFactory);
Future<Void> upload = executor.submit(new Callable<Void>() {
    @Override
    public Void call() throws IOException {
        TransferFsImage.uploadImageFromStorage(activeNNAddress, conf,
            namesystem.getFSImage().getStorage(), imageType, txid, canceler);
        return null;
    }
});
executor.shutdown();
try {
    upload.get();
} catch (InterruptedException e) {
    upload.cancel(true);
    throw e;
} catch (ExecutionException e) {
    throw new IOException("Exception during image upload: " + e.getMessage(),
        e.getCause());
}
}

```

doCheckpoint()方法调用了 FSImage.saveNamespace()方法将当前命名空间保存到新的 fsimage 文件中。saveNamespace()方法首先重置(roll)了 editlog 文件, 将当前的 edit_inprogress 文件关闭并重命名, 为与 fsimage 文件合并做准备。之后调用 saveFSImageInAllDirs()将 fsimage 和 editlog 文件加载到命名空间中, 并将更新的命名空间保存到新的 fsimage 文件中。最后开启新的 editlog_inprogress 文件, 用于记录新的操作。这里涉及各个方法, 在 FSImage 类章节中都已经具体介绍过了, 请读者参考 FSImage 类章节。

Hadoop 2.X HDFS 源码剖析

```
public synchronized void saveNamespace(FSNamesystem source, NameNodeFile nnf,
    Canceler canceler) throws IOException {

    boolean editLogWasOpen = editLog.isSegmentOpen();
    if (editLogWasOpen) {
        // 将当前 edit_inprogress 文件关闭, 并重命名
        editLog.endCurrentLogSegment(true);
    }
    long imageTxId = getLastAppliedOrWrittenTxId();
    try {
        // 调用 saveFSImageInAllDirs() 方法将当前的命名空间保存到新的 fsimage 文件中
        saveFSImageInAllDirs(source, nnf, imageTxId, canceler);
        storage.writeAll();
    } finally {
        if (editLogWasOpen) {
            // 开启新的 editlog_inprogress 文件用于记录请求
            editLog.startLogSegment(imageTxId + 1, true);
            storage.writeTransactionIdFileToStorage(imageTxId + 1);
        }
    }
}
```

将 fsimage 保存成功后, doCheckpoint()方法就会调用 TransferFsImage.uploadImageFromStorage()方法将新生成的 fsimage 文件上传到 Active Namenode 中了, 感兴趣的读者可以自己学习这一部分代码, 这里就不再详细叙述了。

至此, HDFS 中完整的检查点流程就介绍完了。可以看到, 在 HDFS2.X 中引入 HA 机制后, 检查点机制的实现简单了很多, 读者也可以参考 HDFS1.X 与 HDFS2.X 中两种不同的实现, 对比学习与理解 HDFS 的检查点流程。

3.1.5 FSDirectory 类

Namenode 最重要的两个功能之一就是维护整个文件系统的目录树 (即命名空间 namesystem)。HDFS 文件系统的命名空间 (namespace), 也就是以 “/” 为根的整个目录树, 是通过 FSDirectory 类来管理的。这里需要注意的是, FSNamesystem 也提供了管理目录树结构的方法, 但 FSNamesystem 中的方法多是调用 FSDirectory 类的实现, FSNamesystem 在 FSDirectory 类方法的基础上添加了 editlog 日志记录的功能。而 FSDirectory 的操作则全部是在内存中进行的, 并不进行 editlog 的日志记录。

HDFS 之所以引入 FSDirectory, 是想要将文件系统目录树的所有操作抽象成一个统一的接口, 这样其他子系统在调用目录树操作时就不需要了解目录树内部实现的复杂逻辑了, 只需调用 FSDirectory 对象上的指定方法即可。而 FSDirectory 知晓如何将这些目录树的操作分派到具体实现的各个对象上, 并实现对应的逻辑。有了 FSDirectory 这个门面类, 会使得所有对目录树的操作变得简单, 也降低了其他 HDFS 逻辑与目录树逻辑的耦合性。

FSDirectory 的设计使用了门面 (Facade) 模式, 门面模式是指提供一个统一的接口去访问多个子系统的多个不同的接口, 它为子系统中的一组接口提供一个统一的高层接口, 使得子系统更容易使用。

FSDirectory 定义的字段如图 3-33 所示。

① rootDir	INodeDirectory
② namesystem	FSNamesystem
③ skipQuotaCheck	boolean
④ maxComponentLength	int
⑤ maxDirItems	int
⑥ lsLimit	int
⑦ contentCountLimit	int
⑧ inodeMap	INodeMap
⑨ yieldCount	long
⑩ inodeXAttrsLimit	int
⑪ dirLock	ReentrantReadWriteLock
⑫ ezManager	EncryptionZoneManager
⑬ nameCache	NameCache<ByteArray>

图 3-33 FSDirectory 字段

- **rootDir**: 整个文件系统目录树的根节点, 是 INodeDirectory 类型的 (请参考 INode 相关类章节内容)。
- **namesystem**: Namenode 的门面类, 这个类主要支持对数据块进行操作的一些方法, 例如 addBlock()。
- **inodeMap**: 记录根目录下所有的 INode, 并维护 INodeId ->INode 的映射关系。
- **dirLock**: 对目录树以及 inodeMap 字段操作的锁。
- **nameCache**: 将常用的 name 缓存下来, 以降低 byte[] 的使用, 并降低 JVM heap 的使用。

FSDirectory 类中的方法相当多, 主要是封装了对文件系统目录树的操作, 例如增、删、改、查等。基本上 ClientProtocol 中的方法, 都能在 FSDirectory 中找到对应的方法。下面我们介绍 FSDirectory 中几个典型的方法。

1. addChild()

addChild()方法的主要作用是向文件系统目录树中的某个目录节点添加一个 child 子节点, 它在底层调用 INodeDirectory.addChild()方法, 并且更新 Quota 信息以及 FSDirectory.inodeMap 字段。

```
private boolean addChild(INodesInPath iip, int pos,
    INode child, boolean checkQuota) throws QuotaExceededException {

    // ...
    // 检查是否有足够的空间
    if (checkQuota) {
        verifyMaxComponentLength(child.getLocalNameBytes(), inodes, pos);
        verifyMaxDirItems(inodes, pos);
    }
    // 检查 INode 的名称
```

```
verifyINodeName(child.getLocalNameBytes());

final Quota.Counts counts = child.computeQuotaUsage();
updateCount(iip, pos,
    counts.get(Quota.NAMESPACE), counts.get(Quota.DISKSPACE), checkQuota);
final INodeDirectory parent = inodes[pos-1].asDirectory();
boolean added = false;
try {
    // 将 INode 添加到父节点的孩子节点列表中
    added = parent.addChild(child, true, iip.getLatestSnapshot(),
        inodeMap);
} catch (QuotaExceededException e) {
    updateCountNoQuotaCheck(iip, pos,
        -counts.get(Quota.NAMESPACE), -counts.get(Quota.DISKSPACE));
    throw e;
}
if (!added) {
    updateCountNoQuotaCheck(iip, pos,
        -counts.get(Quota.NAMESPACE), -counts.get(Quota.DISKSPACE));
} else {
    iip.setINode(pos - 1, child.getParent());
    addToINodeMap(child);
}
return added;
}
```

2. addBlock()

`addBlock()`方法用于在指定的 `INode` 节点上添加一个数据块，它通过 `namesystem` 获取 `BlockManager` 对象，并在 `BlockManager` 对象上调用 `addBlockCollection()`方法将一个 `Block` 添加到 `BlockManager` 的 `blockMap` 字段中保存，之后调用 `INodeFile.addBlock()`方法将 `Block` 对象添加到 `INode` 对象的 `blocks` 字段中保存。

```
BlockInfo addBlock(String path, INodesInPath inodesInPath, Block block,
    DatanodeDescriptor targets[]) throws IOException {
    waitForReady();

    writeLock();
    try {
        final INodeFileUnderConstruction fileINode =
            INodeFileUnderConstruction.valueOf(inodesInPath.getLastINode(), path);

        // 检查空间是否够用
        updateCount(inodesInPath, 0, fileINode.getBlockDiskspace(), true);

        // 构造 BLock 对象
        BlockInfoUnderConstruction blockInfo =
            new BlockInfoUnderConstruction(
```

```

        block,
        fileINode.getFileReplication(),
        BlockUCState.UNDER_CONSTRUCTION,
        targets);
// 使用 BlockManager 添加 Block
getBlockManager().addBlockCollection(blockInfo, fileINode);
// 将 Block 与文件对应的 INode 对象关联起来
fileINode.addBlock(blockInfo);

return blockInfo;
} finally {
    writeUnlock();
}
}

```

3. setOwner()

setOwner()方法的逻辑非常简单，它在内部调用 unprotectedSetOwner()方法，这里之所以叫作 unprotected，是因为在这个方法内，对于 INode 的修改操作并没有通过 editlog 记录。在本节开头的内容中我们已经学习了 FSDirectory 中所有对于目录树的操作只修改 Namenode 内存中的数据，并不涉及 editlog 记录操作，setOwner()方法的 editlog 记录是由 FSNamesystem.setOwner()方法执行的。unprotectedSetOwner()方法会查找路径上指定文件对应的 INode 对象，然后在这个 INode 对象上调用 setUser()方法更新文件的所属用户。

```

void setOwner(String src, String username, String groupname)
    throws FileNotFoundException, UnresolvedLinkException,
        QuotaExceededException, SnapshotAccessControlException {
    writeLock();
    try {
        unprotectedSetOwner(src, username, groupname);
    } finally {
        writeUnlock();
    }
}

void unprotectedSetOwner(String src, String username, String groupname)
    throws FileNotFoundException, UnresolvedLinkException,
        QuotaExceededException, SnapshotAccessControlException {
    assert hasWriteLock();
    final INodesInPath inodesInPath = getINodesInPath4Write(src, true);
    INode inode = inodesInPath.getLastINode();
    if (inode == null) {
        throw new FileNotFoundException("File does not exist: " + src);
    }
    if (username != null) {
        inode = inode.setUser(username, inodesInPath.getLatestSnapshotId());
    }
    if (groupname != null) {

```

```
inode.setGroup(groupname, inodesInPath.getLatestSnapshotId());  
}  
}
```

上面三个例子简单地描述了 `FSDirectory` 提供的目录树方法，其他方法也基本可以归类为目录树的增、删、改、查等典型操作，实现也都是基于前面章节中讨论过的方法，这里我们就不再多介绍了。在以后的章节中如果遇到重要的 `FSDirectory` 相关的方法，我们会单独进行介绍，其余部分读者可以自行了解。

3.2 数据块管理

如前面的章节所言，`Namenode` 维护着 HDFS 中两个最重要的关系。

- HDFS 文件系统的目录树以及文件的数据块索引。文件系统目录树在 3.1 节中已经介绍过了，文件的数据块索引即每个文件对应的数据块列表，这个信息保存在 `INodeFile.blocks` 字段中，这一部分内容我们在本节中介绍。
- 数据块和数据节点的对应关系，即指定数据块的副本保存在哪些数据节点上的信息。这个信息是在 `Datanode` 启动时，由 `Datanode` 上报给 `Namenode` 的。也就是说，这个信息是 `Namenode` 动态构建起来的，而不是从 `fsimage` 文件中加载的。

在 3.1 节中我们已经详细地介绍了 HDFS 文件系统目录树的实现，我们知道 `Namenode` 会定期将文件系统目录树以及文件与数据块的对应关系保存至 `fsimage` 文件中，然后在 `Namenode` 启动时读取 `fsimage` 文件以重建 HDFS 第一关系。这里要注意的是第二关系并不会保存至 `fsimage` 文件中，也就是说，`fsimage` 并不记录数据块和数据节点的对应关系。这部分数据是由 `Datanode` 主动将当前 `Datanode` 上保存的数据块信息汇报给 `Namenode`，然后由 `Namenode` 更新内存中的数据，以维护数据块和数据节点的对应关系。本节我们就重点学习 `Namenode` 是如何管理内存中的数据块的。

3.2.1 Block、Replica、BlocksMap

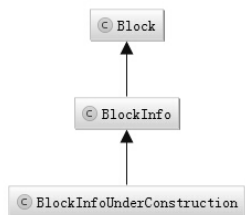


图 3-34 Block 类继承关系图

`INodeFile.blocks` 字段记录了一个 HDFS 文件拥有的所有数据块，也正是通过这个字段 HDFS 第一关系与第二关系发生了关联。我们从 `INodeFile.blocks` 字段入手，学习 HDFS 是如何抽象数据块、如何实现描述数据块和数据节点对应关系的数据结构，以及如何在内存中保存这些数据结构的。

`INodeFile.blocks` 字段是一个 `BlockInfo` 类型的数组，`BlockInfo` 类是 `Block` 的子类，HDFS 使用 `Block` 类抽象 `Namenode` 中的数据块，`Block` 类的继承关系如图 3-34 所示。

1. Block 类

Block 类用来唯一地标识 Namenode 中的数据块，是 HDFS 数据块最基本的抽象接口。Block 类实现了 Writable 接口，是可以序列化的。Block 类还实现了 Comparable 接口，按照 blockid 大小排序。

Block 类定义了三个字段：① blockId 唯一地标识这个 Block 对象；② numBytes 是这个数据块的大小（单位是字节）；③ generationStamp 是这个数据块的时间戳（联想 GenerationStamp 这个类）。

```
private long blockId;
private long numBytes;
private long generationStamp;
```

Block 类定义的方法包括了这三个属性的 get/set()方法、序列化与反序列化方法，以及一些工具方法（例如，isBlockFilename()方法用于检测文件名是否为合法的数据块文件名）。

2. BlockInfo 类

BlockInfo 类扩展自 Block 类，是 Block 类的补充说明。BlockInfo 类定义了 bc 字段保存该数据块归属于哪一个 HDFS 文件，bc 字段是 BlockCollection 类型的，记录了该 HDFS 文件的 INode 对象的引用（INode 是 BlockCollection 的子类）。BlockInfo 类定义了 triplets 字段保存这个 Block 的副本存储在哪些数据节点上，triplets 是一个 Object 类型的数组，这个字段非常重要。

```
private BlockCollection bc;
private Object[] triplets;
```

triplets[]数组是一个比较有趣的数据结构，triplets[]数组有 $3 * replication$ 个元素（replication 是当前数据块拥有的副本个数）。假设 i 为第 i 个保存有该数据块副本的 Datanode，那么 triplets[3*i]是保存这个数据块副本的第 i 个 Datanode 的 DatanodeStorageInfo 对象（DatanodeStorageInfo 是 Namenode 用于描述 Datanode 上存储的对象，请参考数据节点管理小节），triplets[3*i+1]是同一个 Datanode 存储上保存的前一个数据块对应的 BlockInfo 对象，triplets[3*i+2]是同一个 Datanode 存储上保存的后一个数据块对应的 BlockInfo 对象。triplets 这个数据结构构造了一个隐性的双向链表，之所以这样设计 BlockInfo 对象是为了节省内存，同时能够很方便地通过 DatanodeStorageInfo 的引用找到该 Datanode 存储上保存的所有 BlockInfo 对象，因为 DatanodeStorageInfo 持有它存储的第一个数据块对应的 BlockInfo 对象，通过这个 BlockInfo 对象的双向链表就可以获得这个存储上保存的所有数据块。如果使用 LinkedList 双向链表存储这些数据，每个块副本需要 42 个字节，而使用 triplets 只需要 16 个副本，同样完成双向链表功能，却大大节省了内存空间。

这里需要特别注意的是，在 HDFS 2.6 版本前，triplets[3*i]保存的是存储这个数据块副本的第 i 个 Datanode 对应的 DatanodeDescriptor 对象；而在 HDFS 2.6 版本中，将 DatanodeDescriptor 对象更换为 DatanodeStorageInfo 对象，这样做是为了支持 Datanode 的异构存储机制，它将数据块的最小存储单元由 Datanode 更改为 Datanode 上定义的一块 Storage（存

储), 同一个 Datanode 可以定义多个异构的 Storage 来存储数据块。异构存储机制使得 Datanode 可以根据存储数据的类型来使用不同的存储, 例如内存、磁盘、SSD 等, 也就提高了 Datanode 的存储性能。

我们再来看 BlockInfo 中定义的方法。BlockInfo 中定义的方法大都是维护 triplets 这个数据结构的, 例如增加 Datanode 或者修改 prev/next BlockInfo 等。这些方法的实现都比较简单, 读者可以自行阅读。

同时需要注意 convertToBlockUnderConstruction() 方法, 当客户端对数据块进行追加写操作时会调用这个方法, 它将一个 BlockInfo 对象转换为 BlockInfoUnderConstruction 对象, 也就是将该 BlockInfo 对应的数据块状态变为了构建中状态。

```
public BlockInfoUnderConstruction convertToBlockUnderConstruction(
    BlockUCState s, DatanodeStorageInfo[] targets) {
    if(isComplete()) { // 构造 BlockInfoUnderConstruction 对象
        return new BlockInfoUnderConstruction(this,
            getBlockCollection().getBlockReplication(), s, targets);
    }
    // 当前对象已经是一个 BlockInfoUnderConstruction 对象
    BlockInfoUnderConstruction ucBlock = (BlockInfoUnderConstruction)this;
    ucBlock.setBlockUCState(s);
    ucBlock.setExpectedLocations(targets);
    return ucBlock;
}
```

3. BlockInfoUnderConstruction 类

HDFS 在加载 fsimage 时, 如果当前加载的文件处于正在构建状态, 也就是 INodeFile 中包含 FileUnderConstructionFeature 特性, 则将该 INodeFile 的最后一个数据块设置为 BlockInfoUnderConstruction, 表明最后一个数据块正在构建中。这里要注意处于构建状态的 INodeFile 除最后一个数据块为 BlockInfoUnderConstruction 外, 其他均为正常的 BlockInfo。这里我们看一下 FSImageFormatPBINode.loadINodeFile() 方法中构建 BlockInfoUnderConstruction 部分的代码。

```
private INodeFile loadINodeFile(INodeSection.INode n) {
    assert n.getType() == INodeSection.INode.Type.FILE;
    // 从 fsimage 中获取 INodeFile 信息
    INodeSection.INodeFile f = n.getFile();
    //...

    // 如果当前 INodeFile 处于构建状态, 则将最后一个数据块设置为构建状态
    if (f.hasFileUC()) {
        INodeSection.FileUnderConstructionFeature uc = f.getFileUC();
        file.toUnderConstruction(uc.getClientName(), uc.getClientMachine());
        if (blocks.length > 0) {
            BlockInfo lastBlk = file.getLastBlock();
            // 设置最后一个数据块为 BlockInfoUnderConstruction
        }
    }
}
```

```

        file.setBlock(file.numBlocks() - 1, new BlockInfoUnderConstruction(
            lastBlk, replication));
    }
}
return file;
}

```

当 Datanode 汇报当前数据块对应的构建中副本的状态为 FINALIZED，并且当前数据块的状态为 COMMITTD（请参考 Block 类状态与 Replica 类状态小节），且数据块的副本数目大于等于最小副本数目（最小副本数目默认为 1）时，Namenode 会将当前数据块由 BlockInfoUnderConstruction 转变为 BlockInfo，并将数据块状态设置为 COMPLETE。这里需要注意，BlockInfoUnderConstruction 还通过数据结构 replicas 记录了当前数据块对应的所有正在备份的副本的引用。

```
private List<ReplicaUnderConstruction> replicas;
```

4. BlocksMap 类

BlocksMap 是 Namenode 上与数据块相关的最重要的类，它管理着 Namenode 上数据块的元数据，包括当前数据块属于哪个 HDFS 文件，以及当前数据块保存在哪些 Datanode 上。当 Datanode 启动时，会对 Datanode 的本地磁盘进行扫描，并将当前 Datanode 上保存的数据块信息汇报到 Namenode。Namenode 收到 Datanode 的汇报信息后，会建立数据块与保存这个数据块的数据节点的对应关系，并将这个信息保存在 BlocksMap 中。所以无论是获取某个数据块对应的 HDFS 文件，还是获取数据块保存在哪些数据节点上，都需要通过 BlocksMap 对象。

BlocksMap 虽然非常重要，但是它的实现却很简单，它通过一个 GSet 对象维护了 Block -> BlockInfo 的映射关系。GSet 是 Hadoop 自己实现的一个比较特殊的集合类型，其特殊的地方在于它是一个集合，但却提供了类似映射的功能。BlocksMap 定义的字段如下：

```
private final int capacity;
private volatile GSet<Block, BlockInfo> blocks;
```

为什么 BlocksMap 维护的是 Block -> BlockInfo 的对应关系呢？这是因为 BlockInfo 保存数据节点的信息都是在 Datanode 启动时上报的，也就是动态构建的。而 Namenode 启动时内存中保存的关于数据块的信息只有 Block 类中维护的这么多，所以 Namenode 维护了 Block -> BlockInfo 的对应关系，随着 Datanode 不断地上报数据块信息，BlockInfo 的信息会不断地更新。

5. Replica 类状态

在 HDFS 中还有两个比较重要的概念需要我们区分，那就是数据块（block）和副本（replica）。我们将 Namenode 中的数据块信息叫作数据块，但是将 Datanode 中保存的数据块称之为副本。

在 HDFS 源码中使用 Replica 对象描述副本，Replica 类的继承关系如图 3-35 所示。

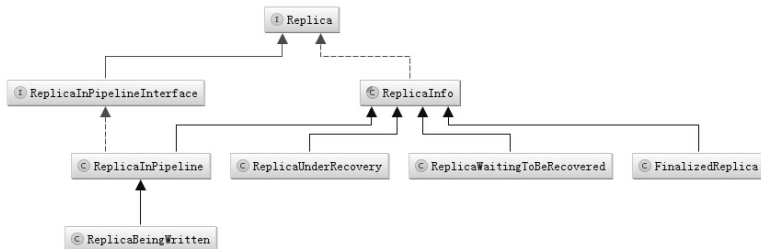


图 3-35 Replica 类继承关系图

HDFS 定义的副本主要有 5 种状态。

```

static public enum ReplicaState {
    FINALIZED(0),
    RBW(1),
    RWR(2),
    RUR(3),
    TEMPORARY(4);
    private int value;
    //...
}

```

- **FINALIZED**: Datanode 上的副本已完成写操作，不再修改。FINALIZED 状态的副本使用 FinalizedReplica 类描述。
- **RBW (ReplicaBeingWritten)**: 刚刚被创建或者追加写的副本，处于写操作的数据流管道中，正在被写入，且已写入副本的内容还是可读的。RBW 状态的副本使用 ReplicaBeingWritten 类描述。
- **RUR (ReplicaUnderRecovery)**: 租约 (Lease) 过期之后发生租约恢复和数据块恢复 (Block recovery, 请参考租约管理中的租约恢复小节) 时副本所处的状态。RUR 状态的副本使用 ReplicaUnderRecovery 类描述。
- **RWR (ReplicaWaitingToBeRecovered)**: 如果一个 Datanode 挂掉并且重启之后，所有 RBW 状态的副本都将转换为 RWR 状态。RWR 状态的副本不会出现在数据流管道中，结果就是等着进行租约恢复操作。RWR 状态的副本使用 ReplicaWaitingToBeRecovered 类描述。
- **TEMPORARY (ReplicaInPipeline)**: Datanode 之间传输副本 (例如 cluster rebalance) 时，正在传输的副本就处于 TEMPORARY 状态。和 RBW 状态的副本不同的是，TEMPORARY 状态的副本内容是不可读的，如果 Datanode 重启，会直接删除处于 TEMPORARY 状态的副本。TEMPORARY 状态的副本使用 ReplicaInPipeline 类描述。

6. Block 类状态

Namenode 中的数据块有以下 4 种状态，注意下面描述中提到的副本状态在上一小节 Replica 类状态中已经介绍过了。

```

static public enum BlockUCState {
    COMPLETE,

```

```

    UNDER_CONSTRUCTION,
    UNDER_RECOVERY,
    COMMITTED;
}

```

- **COMPLETE:** 数据块的 length (长度) 和 gs (时间戳) 不再发生变化, 并且 Namenode 已经收到至少一个 Datanode 报告有 FINALIZED 状态的副本 (replica) (Datanode 上的副本状态发生变化时会通过 blockReceivedAndDeleted() 方法向 Namenode 报告)。一个 COMPLETE 状态的数据块会在 Namenode 的内存中保存所有 FINALIZED 副本的位置。只有当 HDFS 文件的所有数据块都处于 COMPLETE 状态时, 该 HDFS 文件才能被关闭。
- **UNDER_CONSTRUCTION:** 文件被创建或者进行追加写操作时, 正在被写入的数据块就处于 UNDER_CONSTRUCTION 状态。处于该状态的数据块的长度 (length) 和时间戳 (gs) 都是可变的, 但是处于该状态的数据块对于读取操作来说是可见的。
- **UNDER_RECOVERY:** 如果一个文件的最后一个数据块处于 UNDER_CONSTRUCTION 状态时, 客户端异常退出, 该文件的租约 (lease) 超过 softLimit 过期, 该数据块就需要进行租约恢复 (Lease recovery) 和数据块恢复 (Block recovery) 流程释放租约并关闭文件, 那么正在进行租约恢复和数据块恢复流程的数据块就处于 UNDER_RECOVERY 状态。
- **COMMITTED:** 客户端在写文件时, 每次请求新的数据块 (addBlock RPC 请求) 或者关闭文件时, 都会顺带对上一个数据块进行提交 (commit) 操作 (上一个数据块从 UNDER_CONSTRUCTION 状态转换成 COMMITTED 状态)。COMMITTED 状态的数据块表明客户端已经把该数据块的所有数据都发送到了 Datanode 组成的数据流管道 (pipeline) 中, 并且已经收到了下游的 ACK 响应, 但是 Namenode 还没有收到任何一个 Datanode 汇报有 FINALIZED 副本。

3.2.2 数据块副本状态

通过上一节的学习我们知道, Namenode 通过 BlocksMap 维护了数据块副本与数据节点之间的对应关系。在 HDFS 运行时, 一个数据块副本可以存在很多不同的状态, 系统发生异常或者用户执行特定操作时都会对数据块副本状态产生影响。本节就介绍 Namenode 上的数据块副本状态, 以及保存这些不同状态数据块副本的数据结构。

在 HDFS 源码中并没有使用一个枚举类给出数据块副本的状态定义以及状态之间的转移操作, 而是通过 BlockManager 中的数据结构、不同的数据块副本类 (例如 BlockUnderConstruction 和 Block) 以及副本所在 Datanode 的状态 (Datanode 处于撤销状态) 来记录数据块副本的状态。下面我们依次介绍各种情况。

1. BlockManager 数据结构

在 Namenode 的源代码中, 使用 BlockManager 类来管理和维护所有与数据块相关的操作。BlockManager 类方法很多, 类也比较大, 我们在后续的章节中会依次介绍。这里我们首先了

解 BlockManager 中用于保存不同状态数据块副本的数据结构，代码如下：

```
// 损坏的数据块副本集合
final CorruptReplicasMap corruptReplicas = new CorruptReplicasMap();
// 等待删除的数据块副本集合
private final InvalidateBlocks invalidateBlocks;
// 推迟操作的数据块副本集合
private final Set<Block> postponedMisreplicatedBlocks = Sets.newHashSet();
// 多余的数据块副本集合
public final Map<String, LightweightLinkedSet<Block>> excessReplicateMap =
    new TreeMap<String, LightweightLinkedSet<Block>>();
// 等待复制的数据块副本集合
public final UnderReplicatedBlocks neededReplications = new UnderReplicatedBlocks();
// 已经生成复制请求的数据块副本
final PendingReplicationBlocks pendingReplications;
```

- **corruptReplicas:** CorruptReplicasMap 类的实例，保存损坏的数据块副本(corruptReplica)，Datanode 的数据块扫描器发现的错误的数据块副本会放入这个集合中。
- **excessReplicateMap:** 保存多余的数据块副本，当管理员降低 HDFS 文件的副本系数时会产生多余的数据块副本，excessReplicateMap 使用 TreeMap 存放 StorageId (标识 Datanode) -> extraBlocks (多余副本集合) 的映射。
- **invalidateBlocks:** InvalidateBlocks 类的实例，保存等待删除的数据块副本。等待删除的数据块副本来自 corruptReplicas 和 excessReplicateMap 这两个集合，加入这个队列中的数据块副本会由 Namenode 通过名字节点指令向对应的 Datanode 下发删除指令。
- **neededReplications:** UnderReplicatedBlocks 类的实例，保存等待复制的数据块。Namenode 在处理 Datanode 的数据块汇报时，如果检查到数据块的副本数量不够，也就是副本数量小于副本系数时，就会将数据块加入 neededReplications 中存储。当 Namenode 为这个数据块生成复制请求并通过名字节点指令向 Datanode 下发复制命令后，会将这个数据块从 neededReplications 队列中移动到 pendingReplications 中。
- **pendingReplications:** PendingReplicationBlocks 类的实例，保存已经生成复制请求的数据块。当复制请求生成并下发到 Datanode 后，数据块会从 neededReplications 中取出，并放入 pendingReplications 集合中。之所以引入这个数据结构，是为了在数据块复制操作失败后能够进行重试操作。而当数据块复制操作成功后，Datanode 将该数据块副本报告上来，BlockManager 会将数据块从 pendingReplications 队列中删除。
- **postponedMisreplicatedBlocks:** 当 Namenode 发生异常，进行了 Active 与 Standby 切换时，多余的副本不能立即删除，需要先放入 postponedMisreplicatedBlocks 队列中，直到这个数据块的所有副本所在的 Datanode 都进行了块汇报。为什么要有这个数据结构请参考 postponedMisreplicatedBlocks 队列小节中的具体分析。

可以看到，BlockManager 的很多数据结构都没有简单地使用集合类，而是重新定义了新的类作为容器。下面我们就一一分析下上面提到的各个数据结构。

(1) CorruptReplicasMap 类

CorruptReplicasMap 类用于保存损坏的数据块副本(corruptReplica)集合。客户端发现损

坏的数据块时会通过 `ClientProtocol.reportBadBlocks()` 方法向 Namenode 汇报损坏的数据块副本，数据节点会通过 `DatanodeProtocol.reportBadBlocks()` 方法汇报损坏的数据块副本，之后 `BlockManager` 会将损坏的副本加入这个数据结构中。

`CorruptReplicasMap` 保存的是损坏的数据块副本与保存这个副本的 `Datanode` 的对应关系（Block -> Datanode 的映射关系），注意这里同时还保存了这个副本损坏的原因。如下代码所示，`CorruptReplicasMap` 底层使用了一个 `TreeMap`（我们知道 Block 是实现了 `Comparable` 接口的，按照 `blockId` 的大小排序）作为存储的数据结构。

```
// CorruptReplicasMap 底层的 TreeMap 结构
private final SortedMap<Block, Map<DatanodeDescriptor, Reason>> corruptReplicasMap =
    new TreeMap<Block, Map<DatanodeDescriptor, Reason>>();

// 副本损坏的原因
public static enum Reason {
    NONE,                // 没有指明
    ANY,                 // 通配情况
    // Datanode 上副本的时间戳与 Namenode 上数据块的时间戳不一致
    GENSTAMP_MISMATCH,
    // Datanode 上副本的大小与 Namenode 上数据块的大小不一致
    SIZE_MISMATCH,
    INVALID_STATE,       // 无效的状态
    CORRUPTION_REPORTED // 客户端或者数据节点汇报
}
```

（2）InvalidateBlocks 类

`InvalidateBlocks` 类用于保存等待删除的数据块副本集合，`InvalidateBlocks` 中的副本来自于 `corruptReplicas` 和 `excessReplicateMap` 这两个集合。它使用 `TreeMap` 保存了 `Datanode`（使用 `DatanodeInfo` 标识）到该 `Datanode` 上所有等待删除的副本集合的映射。这里使用 `LightWeightHashSet` 对象保存一个 `Datanode` 上所有等待删除的副本集合，`LightWeightHashSet` 是 `Hadoop` 定义的占用较少内存的 `Hashset` 的实现。

```
private final Map<DatanodeInfo, LightWeightHashSet<Block>> node2blocks =
    new TreeMap<DatanodeInfo, LightWeightHashSet<Block>>();
```

`BlockManager` 的 `ReplicationMonitor` 线程（请参考 `BlockManager` 类的 `ReplicationMonitor` 小节）会定期执行删除操作，每次删除时 `ReplicationMonitor` 线程都会从 `InvalidateBlocks` 中选出 `nodeToProcess` 个 `Datanode` 执行删除操作，然后再从每个 `Datanode` 上选出 `limit`（由 `dfs.block.invalidate.limit` 配置项配置，默认值为 1000）个副本删除。由于 `BlockManager` 的删除操作是先选择 `Datanode` 然后选择删除副本，所以 `InvalidateBlocks` 的 `node2blocks` 字段是以 `DatanodeInfo` 作为 key 的，保存了这个 `Datanode` 上所有待删除的副本。

（3）UnderReplicatedBlocks 类

`UnderReplicatedBlocks` 类用于保存所有等待复制的数据块副本集合。和上面两个数据结构相比，`UnderReplicatedBlocks` 的实现相对复杂一点，它维护了一个优先级队列 `priorityQueues`。

`priorityQueues` 是一个列表，它有 5 个子队列，每一个队列对应一个优先级列表，其中 0 为最高优先级，5 为最低优先级。

```
private final List<LightWeightLinkedSet<Block>> priorityQueues
    = new ArrayList<LightWeightLinkedSet<Block>>();
```

那么如何确定待复制数据块的优先级呢？`UnderReplicatedBlocks` 是按照如下规则进行判断的。

- 优先级 0：保存需要立刻备份的数据块。这个数据块只有一个拷贝，或者这个数据块没有活着的拷贝并且有一个拷贝所在的 `Datanode` 正处于离线中。这种类型的数据块很有可能丢失。
- 优先级 1：保存副本数极低的数据块，当实际副本数与期望副本数的比例小于 1:3 时，加入这个队列。
- 优先级 2：保存正处于备份中的数据块，但该数据块的副本数并未达到优先级 1 队列中的比例。
- 优先级 3：数据块的副本数量已经足够，但是数据块副本的分布不是很好，如果一个机架或者交换机宕机很有可能造成数据块完全丢失。
- 优先级 4：保存已经损坏的数据块，就是该数据块对应的所有副本都损坏了。这里的策略是将损坏的数据块放入这个队列中，对没有损坏的副本给予更高的优先级。

`UnderReplicatedBlocks` 的 `getPriority()` 方法用于获取一个指定数据块在 `UnderReplicatedBlocks` 集合中的优先级，它的代码如下：

```
private int getPriority(Block block,
                       int curReplicas,
                       int decommissionedReplicas,
                       int expectedReplicas) {
    assert curReplicas >= 0 : "Negative replicas!";
    if (curReplicas >= expectedReplicas) {
        // 数据块的副本数量足够，但是没有均匀地分布在不同的机架上，返回优先级 3
        return QUEUE_REPLICAS_BADLY_DISTRIBUTED;
    } else if (curReplicas == 0) {
        // 当前数据块没有有效的副本，并且有些副本所在的 Datanode 正处于离线中，优先级 0
        if (decommissionedReplicas > 0) {
            return QUEUE_HIGHEST_PRIORITY;
        }
        // 当前数据块的所有副本都是损坏的，优先级 4
        return QUEUE_WITH_CORRUPT_BLOCKS;
    } else if (curReplicas == 1) {
        // 只有一个有效副本，数据块可能随时丢失，优先级 0
        return QUEUE_HIGHEST_PRIORITY;
    } else if ((curReplicas * 3) < expectedReplicas) {
        // 当前副本数小于期望副本数的三分之一，优先级 1
        return QUEUE_VERY_UNDER_REPLICATED;
    } else {
        // 正常备份状态，优先级 2
    }
}
```



```

        return QUEUE_UNDER_REPLICATED;
    }
}

```

BlockManager 的 ReplicationMonitor 线程(请参考 BlockManager.ReplicationMonitor 小节)会调用 UnderReplicatedBlocks.chooseUnderReplicatedBlocks()方法从 UnderReplicatedBlocks 对象中取出 blocksToProcess 待复制的数据块,然后根据集群的负载情况以及数据块的副本分布,选择一个源数据节点和若干个目标数据节点生成复制请求。chooseUnderReplicatedBlocks()方法每次从 UnderReplicatedBlocks 的优先级队列中选出 blocksToProcess 个待复制副本,然后在 UnderReplicatedBlocks.priorityToReplIdx 字段中记录每个优先级队列的读取偏移量,chooseUnderReplicatedBlocks()方法会在下次调用时从每个优先级队列的读取偏移量位置读取副本。如果最后一个优先级队列的读取偏移量到达了队列的末尾,也就是最近没有新添加的待复制数据块时,则重置所有优先级队列的读取偏移量为 0,然后从优先级最高的优先级队列开始读取操作。chooseUnderReplicatedBlocks()方法的代码如下:

```

public synchronized List<List<Block>> chooseUnderReplicatedBlocks(
    int blocksToProcess) {
    // 初始化返回值列表,保存从每个优先级队列中取出的数据块
    List<List<Block>> blocksToReplicate = new ArrayList<List<Block>>(LEVEL);
    for (int i = 0; i < LEVEL; i++) {
        blocksToReplicate.add(new ArrayList<Block>());
    }

    if (size() == 0) { // UnderReplicatedBlocks 没有保存任何待复制数据块
        return blocksToReplicate;
    }

    int blockCount = 0;
    // 遍历 UnderReplicatedBlocks 中的所有优先级队列
    for (int priority = 0; priority < LEVEL; priority++) {
        // 当前优先级队列保存的待复制数据块的迭代器
        BlockIterator neededReplicationsIterator = iterator(priority);
        // 获取当前优先级队列的读取偏移量
        Integer replIndex = priorityToReplIdx.get(priority);

        //从 priorityToReplIdx 字段记录的读取游标开始读取数据
        for (int i = 0; i < replIndex && neededReplicationsIterator.hasNext(); i++) {
            neededReplicationsIterator.next();
        }
        // 获取从当前队列中读取的副本数量
        blocksToProcess = Math.min(blocksToProcess, size());

        if (blockCount == blocksToProcess) {
            break; // 如果读取了足够的数量,则退出循环
        }

        // 读取副本,并将副本保存到 blocksToReplicate 返回值列表中
    }
}

```

```
while (blockCount < blocksToProcess
    && neededReplicationsIterator.hasNext()) {
    Block block = neededReplicationsIterator.next();
    blocksToReplicate.get(priority).add(block);
    replIndex++;
    blockCount++;
}

if (!neededReplicationsIterator.hasNext()
    && neededReplicationsIterator.getPriority() == LEVEL - 1) {
    // 将所有优先级队列的读取偏移量重置为 0，因为最近没有新添加的待复制副本
    for (int i = 0; i < LEVEL; i++) {
        priorityToReplIdx.put(i, 0);
    }
    break;
}
// 更新当前队列的读取游标
priorityToReplIdx.put(priority, replIndex);
}
return blocksToReplicate; // 返回所有待复制数据块
}
```

当 **ReplicationMonitor** 线程为获取的待复制数据块生成了名字节点指令，并且待复制数据块的副本系数已经满足时，**ReplicationMonitor** 会调用 **UnderReplicatedBlocks.remove()**方法将这个数据块从 **UnderReplicatedBlocks** 对象中移除，同时由于删除元素的操作会造成 **UnderReplicatedBlocks.priorityToReplIdx** 字段中记录的队列读取偏移量错误，**BlockManager** 会在调用完 **remove()**方法后，调用 **decrementReplicationIndex()**方法更新这个数据块所在优先级队列的读取偏移量。**remove()**方法和 **decrementReplicationIndex()**方法的代码如下：

```
boolean remove(Block block, int priLevel) {
    if(priLevel >= 0 && priLevel < LEVEL
        && priorityQueues.get(priLevel).remove(block)) { // 从指定队列删除数据块
        return true;
    } else {
        for (int i = 0; i < LEVEL; i++) {
            if (priorityQueues.get(i).remove(block)) { // 遍历所有队列，查找删除数据块
                return true;
            }
        }
    }
    return false;
}

// 更新 priority 队列的读取偏移量
public void decrementReplicationIndex(int priority) {
    Integer replIdx = priorityToReplIdx.get(priority);
    priorityToReplIdx.put(priority, --replIdx); // 将 priority 队列的读取偏移量减 1
}
```

(4) PendingReplicationBlocks 类

PendingReplicationBlocks 类用于存放已经生成复制请求的数据块副本。之所以有这个数据结构，是因为虽然 UnderReplicatedBlocks 队列中已经保存了要复制的数据块副本，但是在数据块复制过程中很有可能出现错误。所以需要将已经生成复制请求的数据块副本放入 PendingReplicationBlocks 类中缓存，如果出现复制失败的情况，则将该数据块副本重新加入 UnderReplicatedBlocks 队列中即可。

PendingReplicationBlocks 类的 pendingReplications 字段保存了数据块 (Block) 到数据块复制信息 (PendingBlockInfo) 的映射关系。PendingBlockInfo 对象中保存了最近一次复制操作的时间戳，以及正在对当前数据块进行复制操作的数据节点。

```
private final Map<Block, PendingBlockInfo> pendingReplications;

static class PendingBlockInfo {
    private long timeStamp;
    private final List<DatanodeDescriptor> targets;
}
```

数据块副本的复制操作执行成功后，也就是 Datanode 成功地保存了数据块的一个副本时，Datanode 会通过增量块汇报接口 DatanodeProtocol.blockReceivedAndDeleted() 通知 BlockManager 对象。BlockManager 会调用 addBlock() 方法将这个新添加的副本信息加入内存中，同时由于副本已经成功地写入数据节点了，所以 addBlock() 方法会调用 PendingReplicationBlocks.decrement() 方法从 pendingReplications 队列中删除该数据节点上的复制请求。decrement() 方法会从 pendingReplications 字段中提取数据块副本对应的 PendingBlockInfo 对象，然后从这个 PendingBlockInfo 中删除汇报的 Datanode 的复制请求。如果该数据块副本对应的 PendingBlockInfo 中不再有任何复制请求，则将这个数据块从 pendingReplications 字段中删除。decrement() 方法的代码如下：

```
void decrement(Block block, DatanodeDescriptor dn) {
    synchronized (pendingReplications) {
        PendingBlockInfo found = pendingReplications.get(block);
        if (found != null) {
            found.decrementReplicas(dn);
            if (found.getNumReplicas() <= 0) {
                pendingReplications.remove(block);
            }
        }
    }
}
```

如果复制操作没有成功，则复制请求会一直保存在 pendingReplications 字段中，直到复制请求过期被放入 timedOutItems 字段中。PendingReplicationBlocks 类会启动一个线程 (timerThread 字段保存) 定期检查 pendingReplications 字段中保存的复制请求，标识那些已经超时的请求并将超时请求放入超时队列 timedOutItems 中。PendingReplicationBlocks 定义的字段如下：

Hadoop 2.X HDFS 源码剖析

```
class PendingReplicationBlocks {
    private static final Log LOG = BlockManager.LOG;

    private final Map<Block, PendingBlockInfo> pendingReplications; // 所有复制请求
    private final ArrayList<Block> timedOutItems; // 过期的复制请求
    Daemon timerThread = null; // 超时检查线程
    private volatile boolean fsRunning = true;

    private long timeout = 5 * 60 * 1000; // 复制请求超时时间
    private final static long DEFAULT_RECHECK_INTERVAL = 5 * 60 * 1000; // 扫描周期
}
```

PendingReplicationBlocks.timerThread 检查线程是由 PendingReplicationMonitor 类实现的，这个线程是在 BlockManager.active()方法中启动的。PendingReplicationMonitor 线程会定期调用 pendingReplicationCheck()方法检查 pendingReplications 集合中的过期请求，这里的检查间隔是由 timeout 字段和 DEFAULT_RECHECK_INTERVAL 常量共同决定的，它们的默认值都是 5 分钟，Math.min(DEFAULT_RECHECK_INTERVAL, timeout)决定了 timerThread 线程多久检查一次所有的复制请求。PendingReplicationBlocks.run()方法的代码如下：

```
public void run() {
    while (fsRunning) {
        long period = Math.min(DEFAULT_RECHECK_INTERVAL, timeout);
        try {
            pendingReplicationCheck();
            Thread.sleep(period);
        } catch (InterruptedException ie) {
        }
    }
}
```

PendingReplicationMonitor.pendingReplicationCheck()方法会判断如果数据块复制请求的请求发起时间和当前时间间隔大于 timeout（默认为 5 分钟），则标识该复制请求为过期请求，并将这个请求放入 timedOutItems 队列中。所有 timedOutItems 中保存的过期请求都会被 BlockManager 重新插入到 UnderReplicatedBlocks 对象中，以重新执行复制操作。pendingReplicationCheck()方法的代码如下：

```
void pendingReplicationCheck() {
    synchronized (pendingReplications) {
        Iterator<Map.Entry<Block, PendingBlockInfo>> iter =
            pendingReplications.entrySet().iterator();

        long now = now();
        while (iter.hasNext()) { // 迭代判断 pendingReplications 中保存的所有请求
            Map.Entry<Block, PendingBlockInfo> entry = iter.next();
            PendingBlockInfo pendingBlock = entry.getValue();
            if (now > pendingBlock.getTimeStamp() + timeout) {
                Block block = entry.getKey();
                synchronized (timedOutItems) {
```

```

        timedOutItems.add(block); // 如果超时则放入 timedOutItems 中保存
    }
    iter.remove();
}
}
}
}
}
}
}

```

(5) postponedMisreplicatedBlocks 队列

当 Namenode 发生错误并进行了 Active 与 Standby 切换时, Namenode 中保存的多余副本不能直接被删除, 需要先放入 postponedMisreplicatedBlocks 队列中, 直到这个数据块的所有副本所在的 Datanode 都进行了块汇报。

为什么要这样设计呢? 是考虑到下面这个场景。

- blockA 有两个副本, 副本 1 在 datanode1 上, 副本 2 在 datanode2 上。
- 这时, namenode1 发出删除指令, 删除 datanode1 上的副本 1。
- 发出删除指令后, namenode1 发生错误, 切换至 namenode2。
- 这时 datanode1 并没有进行块汇报, namenode2 并不知道 datanode1 上已经删除了副本 1。所以 namenode2 向 datanode2 发出删除操作。
- datanode2 删除副本 2, 数据块的所有副本都被删除了, 数据块也就丢失了。

为了解决上述问题, 当 Namenode 发生 Active -> Standby 切换时, 会将所有 Datanode 设置为 stale 状态。当一个 Datanode 处于 stale 状态时, 这个 Datanode 上的所有副本被设置为 stale 状态, 如果数据块有至少一个 stale 状态的副本, 那么这个数据块会被标识为 stale 状态。对于 stale 状态的数据块是不可以因为超过副本系数而直接删除的, 需要先放入 postponedMisreplicatedBlocks 中。当 stale 状态的 Datanode 进行了块汇报操作之后, Namenode 会重新扫描 postponedMisreplicatedBlocks 队列中的数据块, 只有当数据块对应的所有副本都不再处于 stale 状态时, 才可以执行数据块副本的删除操作。这里参考 BlockManager.processMisReplicatedBlock()方法的实现:

```

if (numCurrentReplica > expectedReplication) {
    if (num.replicasOnStaleNodes() > 0) {
        // 副本数多余, 并且存在 stale 的副本, 则放入 POSTPONE 队列
        return MisReplicationResult.POSTPONE;
    }
    // ...
}

```

2. 数据块副本状态 (done)

了解了 BlockManager 中定义的数据结构, 我们可以将 BlockManager 中不同数据结构保存的数据块副本分为如下几种状态, 如图 3-36 所示。

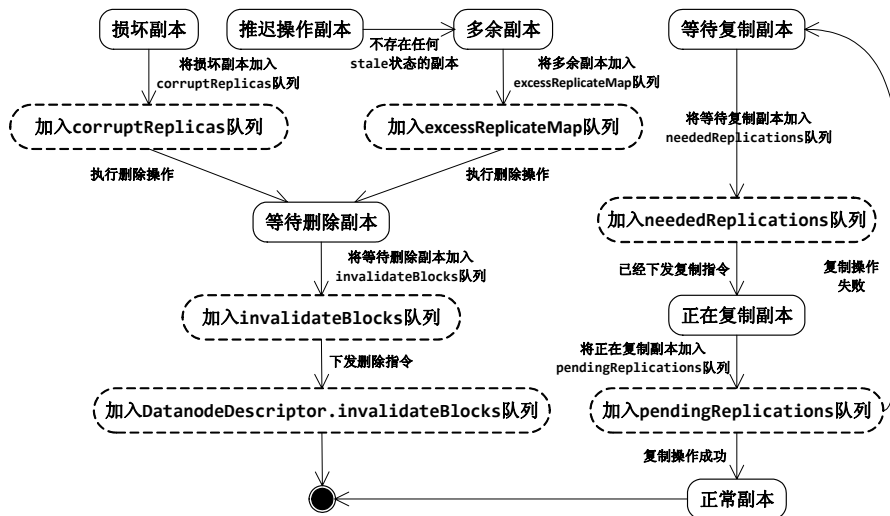


图 3-36 数据块副本状态转换图

- 正常副本：保存在 `BlockManager.BlocksMap` 集合中的正常的数据块副本。
- 损坏副本：存放在 `corruptReplicas` 队列中的损坏的数据块副本。损坏的副本一般由 `Datanode` 的数据块扫描器、数据块接收器或者客户端的数据块校验过程发现。损坏状态的副本会转换为等待删除状态的副本。
- 多余副本：存放在 `excessReplicateMap` 中的数据块副本，也就是副本数目大于副本系数的数据块副本，这种情况可能发生在更改数据块的副本系数之后。多余状态的副本会转换为等待删除状态的副本。
- 等待删除副本：存放在 `invalidateBlocks` 中的数据块副本，也就是等待删除的副本，需要由 `Namenode` 向存放副本的 `Datanode` 下发删除指令。
- 等待复制副本：存放在 `neededReplications` 中等待进行复制操作的数据块副本，但是 `Namenode` 还没有生成复制命令，等待复制状态的副本会转换为正在复制状态的副本。
- 正在复制副本：`Namenode` 已生成复制命令并下发到 `Datanode`，但 `Namenode` 当前并不知道副本复制操作是否成功。如果复制动作成功，那么正在复制状态的副本就会转换为正常状态的副本；如果复制失败，则状态重新转换至等待复制状态。
- 推迟操作的副本：副本数目过多，但是存在 `stale` 副本，所以推迟处理，等待 `stale` 状态的 `Datanode` 进行块汇报。如果当前数据块不存在任何 `stale` 状态的副本，则状态可以转换为多余副本状态，进行删除操作。

同时我们知道，在客户端进行写数据块或者追加写数据块操作时，会出现正在构建状态的数据块副本以及恢复状态的数据块副本。

- 正在构建状态：当前数据块副本正在构建当中，也就是有客户端正在写这个数据块副本，或者追加写这个数据块副本时，数据块副本的状态就为正在构建状态。当数据块完成了写操作后，数据块副本恢复为正常状态。

- 恢复状态：当客户端在写数据块副本以及追加写数据块副本时，客户端崩溃造成了租约过期，则 Namenode 会触发租约恢复操作（请参考租约管理中的租约恢复小节），正在进行租约恢复操作的数据块副本就处于恢复状态。

当 Datanode 被撤销时（当前 Datanode 离线，或者放入 exclude 文件中时），正在撤销的 Datanode 或者已经撤销的 Datanode 上的副本会处于“撤销节点”状态，这些副本由于 Datanode 撤销而进入“最终状态”，或者由于节点重新进入工作状态而返回“正常副本”状态。

3.2.3 BlockManager 类（done）

前面的两个小节介绍了 Block 类、Replica 类和 BlocksMap 类的实现，以及 Namenode 中数据块副本状态之间的转换。本节开始正式介绍 BlockManager 类的实现，BlockManager 类保存并且管理了 HDFS 集群中所有数据块的元数据。我们依然遵循之前的原则，先从入口方法开始介绍。

1. ReplicationMonitor

从 BlockManager 的 activate() 方法我们可以看到，BlockManager 在初始化时启动了两个线程，即 pendingReplications 和 replicationThread，这两个线程共同完成了 BlockManager 的核心逻辑。

```
public void activate(Configuration conf) {
    pendingReplications.start();
    datanodeManager.activate(conf);
    this.replicationThread.start();
}
```

这里的 pendingReplications 线程，我们在 PendingReplicationBlocks 类小节中已经介绍过了，就是 PendingReplicationBlocks.timerThread 线程，由 PendingReplicationMonitor 类实现，这个线程会遍历 pendingReplications 集合中保存的所有数据块复制任务，将超过指定时间（5 分钟）没有确认的复制请求加入超时队列 PendingReplicationBlocks.timedOutItems 中。

再来看一下 replicationThread 线程。replicationThread 的实现类是 BlockManager 中的内部类 ReplicationMonitor，ReplicationMonitor 是一个线程类。ReplicationMonitor 线程会周期性地调用 computeDatanodeWork() 方法触发数据块的复制和删除任务，然后调用 processPendingReplications() 方法将 PendingReplicationBlocks.timedOutItems 队列中保存的超时任务重新加回 neededReplications 队列中。ReplicationMonitor.run() 方法的代码如下：

```
public void run() {
    while (namesystem.isRunning()) {
        try {
            computeDatanodeWork();
            processPendingReplications();
            Thread.sleep(replicationRecheckInterval);
        } catch (Throwable t) {
```

```
        //... 异常处理省略
    }
}
}
```

下面我们来看一下 `computeDatanodeWork()` 和 `processPendingReplications()` 两个方法的实现。

(1) `computeDatanodeWork()`

`computeDatanodeWork()` 方法执行了两项操作。

- 复制操作：从等待复制的数据块中选出若干个数据块执行复制操作，然后为这些数据块的复制操作选出 `source` 源节点以及 `target` 目标节点，最后构造复制指令并在下次心跳时将复制指令带回给源节点以执行复制操作。
- 删除操作：从等待删除的数据块副本中选出若干个副本，然后构造删除指令，并在下次心跳时将删除指令带到目标节点以执行副本的删除操作。

`computeDatanodeWork()` 方法还有一点需要注意的是，对于数据块的复制操作，每次复制的数据块数量为集群中 `Datanode` 数量的 `blocksReplWorkMultiplier` 倍（由配置项 `dfs.namenode.replication.work.multiplier.per.iteration` 配置，默认为 2）。例如集群中有 100 个节点，`Replication Monitor` 在一个周期中只会从 `neededReplications` 集合中取出 200 (2×100) 个数据块进行复制操作。HDFS 之所以这样设计，是考虑到如果一次冗余复制过多的数据块，则会造成 HDFS 集群的网络拥塞，所以需要根据 `Datanode` 的数量来决定进行复制操作的数据块的数量。而对于数据块的删除操作，每次进行删除操作的 `Datanode` 数量占集群中 `Datanode` 数量的百分比为 `blocksInvalidateWorkPct`（由配置项 `dfs.namenode.invalidate.work.pct.per.iteration` 配置，默认为 32%），而每个进行删除操作的 `Datanode` 最多可以删除 `blockInvalidateLimit` 个副本（由配置项 `dfs.block.invalidate.limit` 配置，默认为 1000）。例如集群中有 100 个节点，`ReplicationMonitor` 在一个周期中只会从 32 个 (100×0.32) `Datanode` 上删除数据块，而每个 `Datanode` 上最多可删除 1000 个数据块，也就是总共可以删除 32000 个数据块。

读者可能会问，相对于复制操作，为什么删除操作可以处理更多的数据块呢？这是因为数据块的复制操作会涉及数据块在 `Datanode` 间的移动，会占用集群的网络资源，如果每次复制的数据块太多，则会造成集群的网络拥塞，影响整个 HDFS 集群的性能；而对于删除操作，只需将数据块从 `Datanode` 上删除即可，无须占用其他资源。

`computeDatanodeWork()` 方法的代码如下所示，它分别调用了 `computeReplicationWork()` 方法来执行复制操作，以及 `computeInvalidateWork()` 方法来执行删除操作。复制的副本数量由变量 `blocksToProcess` 指定，进行删除操作的 `Datanode` 数量则由变量 `nodesToProcess` 指定。

```
int computeDatanodeWork() {
    // 处于安全模式下不可以进行复制以及删除操作
    if (namesystem.isInSafeMode()) {
        return 0;
    }
}
```



```

// 获取集群中所有有效的 Datanode 的数量
final int numlive = heartbeatManager.getLiveDatanodeCount();
// 计算出进行复制操作的数据块数量
final int blocksToProcess = numlive * this.blocksReplWorkMultiplier;
// 计算出进行删除操作的 Datanode 数量
final int nodesToProcess = (int) Math.ceil(numlive * this.blocksInvalidateWorkPct);

// 调用 computeReplicationWork() 计算出需要进行备份的副本
int workFound = this.computeReplicationWork(blocksToProcess);

// ...

// 调用 computeInvalidateWork() 计算出需要进行删除的副本
workFound += this.computeInvalidateWork(nodesToProcess);
return workFound;
}

```

下面我们分别学习 `computeReplicationWork()` 和 `computeInvalidateWork()` 方法的实现。

computeReplicationWork()

`computeReplicationWork()` 方法的逻辑如图 3-37 所示, 它会先从 `needReplications` 队列中选出 `blocksToProcess` 个需要冗余复制的数据块, 然后为这些数据块选择源节点 `source` 以及目标节点 `target`, 最后为数据块生成名字节点指令并通过下一次心跳带回到源节点 `source`。

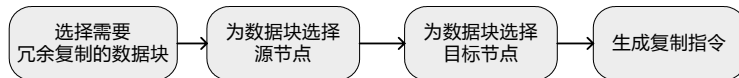


图 3-37 `computeReplicationWork()` 方法逻辑

了解了 `computeReplicationWork()` 的执行逻辑后, 我们来看这个方法的代码实现。`computeReplicationWork()` 方法会首先调用 `needReplications.chooseUnderReplicatedBlocks()` 方法从 `needReplications` 队列中选出 `blocksToProcess` 个需要进行备份的副本 (`chooseUnderReplicatedBlocks()` 方法的实现请参考 `UnderReplicatedBlocks` 类小节)。之后调用 `computeReplicationWorkForBlocks()` 方法执行数据块复制操作。

```

int computeReplicationWork(int blocksToProcess) {
    List<List<Block>> blocksToReplicate = null;
    namesystem.writeLock();
    try {
        // 选出需要进行备份的副本集合
        blocksToReplicate = neededReplications
            .chooseUnderReplicatedBlocks(blocksToProcess);
    } finally {
        namesystem.writeUnlock();
    }
    return computeReplicationWorkForBlocks(blocksToReplicate);
}

```

`chooseUnderReplicatedBlocks()`方法的实现比较简单,就是从 `needReplications` 优先级队列中取出 `blockToProcess` 个元素,并按照优先级放入 `blocksToReplicate` 返回值列表中,在 `UnderReplicatedBlocks` 类小节我们已经详细介绍过这个方法了,这里不再赘述。我们重点看一下 `computeReplicationWorkForBlocks()`的实现,这个方法比较长,我们将它的实现分为三个部分来学习:① 选择源节点;② 选择目标节点;③ 进行复制操作。

选择源节点——`computeReplicationWorkForBlocks()`这部分代码会为数据块的复制操作选择源节点,并确定需要备份的副本数量。在选择源节点操作时,有一个非常重要的方法 `chooseSourceDatanode()`,这个方法会计算出当前数据块的有效备份数、无效备份数以及超出的备份数,同时选择一个源节点作为数据块备份的数据源。`chooseSourceDatanode()`方法选择源节点的策略很简单,如果当前节点处于 `DECOMMISSION_INPROGRESS` 状态,也就是退役节点,那么就选择当前节点,因为退役节点没有写操作占用带宽。其他情况时,随机选择一个节点就好。需要注意的是,如果一个节点上已经有两个或者两个以上的数据块复制任务了,则选择其他节点。

```
int computeReplicationWorkForBlocks(List<List<Block>> blocksToReplicate) {
    // ...

    // 选择源节点代码
    for (int priority = 0; priority < blocksToReplicate.size(); priority++) {
        for (Block block : blocksToReplicate.get(priority)) {
            // 通过 blocksMap 获得该 block 所属的 INode 对象
            bc = blocksMap.getBlockCollection(block);
            // 如果文件正在构建当中,则不可以备份,从 neededReplications 中删除
            if(bc == null || bc instanceof MutableBlockCollection) {
                neededReplications.remove(block,priority);
                neededReplications.decrementReplicationIndex(priority);
                continue;
            }
            // 数据块的副本系数
            requiredReplication = bc.getBlockReplication();

            // 调用 chooseSourceDatanode() 获取备份副本的源节点
            containingNodes = new ArrayList<DatanodeDescriptor>();
            liveReplicaNodes = new ArrayList<DatanodeDescriptor>();
            NumberReplicas numReplicas = new NumberReplicas();
            srcNode = chooseSourceDatanode(
                block, containingNodes, liveReplicaNodes, numReplicas,
                priority);

            // 当前有效的副本数 = 已经备份的数目 + 正在备份的副本
            numEffectiveReplicas = numReplicas.liveReplicas() +
                pendingReplications.getNumReplicas(block);

            // 如果副本数已经够了,则不用进行备份操作,从 neededReplications 队列中删除
            if (numEffectiveReplicas >= requiredReplication) {
```

```

        if ( (pendingReplications.getNumReplicas(block) > 0) ||
            (blockHasEnoughRacks(block)) ) {
            neededReplications.remove(block,priority);
            neededReplications.decrementReplicationIndex(priority);
        }
    }

    // 如果副本数不足, 则计算需要添加多少个副本
    if (numReplicas.liveReplicas() < requiredReplication) {
        additionalReplRequired = requiredReplication
            - numEffectiveReplicas;
    } else {
        additionalReplRequired = 1; // 虽然副本数足够, 但是分布在同一个机架上
    }

    // 在工作队列中添加备份任务, 任务会在下一次心跳时带到 Datanode
    work.add(new ReplicationWork(block, bc, srcNode,
        containingNodes, liveReplicaNodes, additionalReplRequired,
        priority));
}

// 选择目标节点...
// 执行复制操作...
}

```

选择目标节点——选择完源节点后, `computeReplicationWorkForBlocks()` 方法会调用 `chooseTarget()` 方法选择目标节点来保存这个新的数据块副本。`chooseTarget()` 方法的策略依然是机架感知法, 它保证数据块的第一个副本放在本机架或者同机架的 `Datanode` 上; 第二个副本放在不同机架的 `Datanode` 上; 第三个副本放在与第二个副本同机架的另一个 `Datanode` 上。

```

int computeReplicationWorkForBlocks(List<List<Block>> blocksToReplicate) {
    // ...选择源节点代码

    // 选择目标节点代码
    final Set<Node> excludedNodes = new HashSet<Node>();
    for(ReplicationWork rw : work){
        // 排除数据块已有副本所在的 Datanode
        excludedNodes.clear();
        for (DatanodeDescriptor dn : rw.containingNodes) {
            excludedNodes.add(dn);
        }

        // 调用 chooseTarget() 方法, 选择目标节点
        rw.chooseTargets(blockplacement, storagePolicySuite, excludedNodes);
    }

    // ...执行复制操作代码
}

```

进行复制操作——`computeReplicationWorkForBlocks()`完成了源节点和目标节点的选择操作后，就会通过名字节点指令向目标 `Datanode` 发送复制指令，复制指令会通过下一次心跳传递给源节点。之后 `computeReplicationWorkForBlocks()`方法会将已经发送了复制指令的数据块放入 `pendingReplications` 队列中保存，并且在数据块的副本数目满足副本系数时从 `neededReplications` 队列中移除当前数据块。

```
int computeReplicationWorkForBlocks(List<List<Block>> blocksToReplicate) {
    // ...选择源节点代码
    // ...选择目标节点代码

    // 将副本加入 Datanode 的备份队列中，在下次心跳时，发出备份指令
    rw.srcNode.addBlockToBeReplicated(block, targets);
    scheduledWork++;

    for (DatanodeDescriptor dn : targets) {
        dn.incBlocksScheduled();
    }

    // 加入 pendingReplications 队列中
    pendingReplications.increment(block, targets);

    // 如果副本数目足够，则从 neededReplications 队列中删除
    if(numEffectiveReplicas + targets.length >= requiredReplication) {
        neededReplications.remove(block, priority);
        neededReplications.decrementReplicationIndex(priority);
    }

    // ...
}
```

介绍完 `computeReplicationWorkForBlocks()`方法的实现后，我们来看一下 `Namenode` 是如何将数据块的复制指令发送给 `Datanode` 的。`computeReplicationWorkForBlocks()`方法会调用 `DatanodeDescriptor.addBlockToBeReplicated()`方法将待复制的数据块加入源节点对应的 `DatanodeDescriptor` 对象的 `replicateBlocks` 队列中保存，`BlockManager` 会在下次处理这个 `Datanode` 的心跳时从 `DatanodeDescriptor.replicateBlocks` 队列中取出所有待复制的数据块，然后生成名字节点指令并通过心跳的响应带回给源 `Datanode`。`Datanode` 收到心跳带回的复制指令之后，会执行数据块的复制操作，完成数据块的复制操作之后，目标 `Datanode` 会通过增量块汇报接口通知 `Namenode` 数据块已经成功复制了。

`computeInvalidateWork()`

`computeInvalidateWork()`方法的逻辑如图 3-38 所示，它会先从 `BlockManager.invalidateBlocks` 队列中选出 `nodesToProcess` 个 `Datanode` 执行删除副本的操作，然后在每个 `Datanode` 上选择 `blockInvalidateLimit`（由配置项 `dfs.block.invalidate.limit` 配置，默认为 1000）个副本删除。完成了副本的选择操作后，`computeInvalidateWork()`会为待删除的副本生成删除指令，然后通过 `Datanode` 的心跳响应将删除指令带回 `Datanode` 节点。

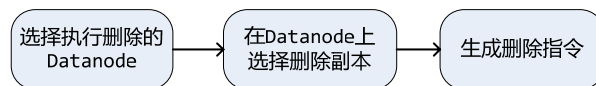


图 3-38 computeInvalidateWork()方法逻辑

了解了 computeInvalidateWork() 的执行逻辑后，我们来看这个方法的代码实现。computeInvalidateWork() 方法首先会调用 InvalidateBlocks.getDatanodes() 方法从 BlockManager.invalidateBlocks 队列中随机选出 nodesToProcess 个 Datanode，然后调用 invalidateWorkForOneNode() 方法删除指定 Datanode 上的无效数据块。

```

int computeInvalidateWork(int nodesToProcess) {
    // 从 invalidateBlocks 中选出所有存在无效数据块的 Datanode
    final List<DatanodeInfo> nodes = invalidateBlocks.getDatanodes();
    Collections.shuffle(nodes);
    // 处理 nodesToProcess 个数据节点
    nodesToProcess = Math.min(nodes.size(), nodesToProcess);

    int blockCnt = 0;
    for (DatanodeInfo dnInfo : nodes) { // 遍历这些数据节点
        // 对于每个数据节点调用 invalidateWorkForOneNode() 方法删除无效数据块
        int blocks = invalidateWorkForOneNode(dnInfo);
        if (blocks > 0) {
            blockCnt += blocks;
            if (--nodesToProcess == 0) {
                break;
            }
        }
    }
    return blockCnt;
}

```

invalidateWorkForOneNode() 方法会调用 invalidateBlocks.invalidateWork() 方法执行单个 Datanode 的删除逻辑，我们只需看一下 InvalidateBlocks.invalidateWork() 方法的实现就好 (InvalidateBlocks 类请参考 InvalidateBlocks 类小节的说明)。这个方法很简单，它直接从 InvalidateBlocks.node2blocks 字段保存的 Datanode 节点对应的待删除数据块副本集合中取出 limit 个元素，然后发出删除指令。要注意每个数据节点一次删除操作只可以删除 limit 个数据块，limit 由 dfs.block.invalidate.limit 配置项配置，默认值为 1000，这样设计的原因是为了保证 Datanode 的处理效率。

```

private synchronized List<Block> invalidateWork(
    final String storageId, final DatanodeDescriptor dn) {
    // 从 InvalidateBlocks.node2blocks 数据结构中取出数据节点对应的待删除副本集合
    final LightweightHashSet<Block> set = node2blocks.get(storageId);
    if (set == null) {
        return null;
    }
}

```

```
// 每次心跳所能发送的指令是有限制的
final int limit = datanodeManager.blockInvalidateLimit;
final List<Block> toInvalidate = set.pollN(limit);

// 如果当前节点上没有需要删除的副本，则删除数据结构中节点的入口
if (set.isEmpty()) {
    remove(storageId);
}

// 在 Datanode 中保存需要删除的副本，在下次心跳时发出删除指令
dn.addBlocksToBeInvalidated(toInvalidate);
numBlocks -= toInvalidate.size();
return toInvalidate;
}
```

与数据块的复制操作一样，每个 `Datanode` 对应的 `DatanodeDescriptor` 对象都有一个 `invalidateBlocks` 队列用于存储需要删除的数据块。`invalidateWork()` 方法取出待删除的副本后，会调用 `DatanodeDescriptor.addBlocksToBeInvalidated()` 方法将这些副本加入 `DatanodeDescriptor.invalidateBlocks` 队列中。`BlockManager` 在这个 `Datanode` 的下次心跳处理时会提取出 `invalidateBlocks` 队列中保存的副本，生成删除副本的名字节点指令，并在下次心跳响应时将删除指令传回 `Datanode`。`Datanode` 接收到心跳响应后，会提取出副本删除指令并执行删除操作，`Datanode` 上的数据块删除操作完成后，`Datanode` 会通过增量块汇报接口通知 `Namenode` 数据块已经成功删除。

(2) processPendingReplications()

`ReplicationMonitor.run()` 调用完 `computeDatanodeWork()` 方法后，会调用 `processPendingReplications()` 方法将 `pendingReplications` 队列中记录的超时的副本重新加入 `neededReplication` 队列（请参考 `PendingReplicationBlocks` 类小节的介绍）。

`processPendingReplications()` 方法会首先调用 `PendingReplicationBlocks.getTimedOutBlocks()` 方法取出已经超时的备份请求，然后判断每个要求进行备份操作的数据块是否具备冗余备份条件，如果满足备份条件，则将这个数据块重新加入 `neededReplication` 队列中。`processPendingReplications()` 方法的代码如下：

```
private void processPendingReplications() {
    Block[] timedOutItems = pendingReplications.getTimedOutBlocks();
    if (timedOutItems != null) {
        namesystem.writeLock();
        try {
            for (int i = 0; i < timedOutItems.length; i++) {
                NumberReplicas num = countNodes(timedOutItems[i]);
                if (isNeededReplication(timedOutItems[i], getReplication(timedOutItems[i]),
                    num.liveReplicas())) {
                    neededReplications.add(timedOutItems[i],
                        num.liveReplicas(),
                        num.decommissionedReplicas(),
```

```

        getReplication(timedOutItems[i]));
    }
} finally {
    namesystem.writeUnlock();
}
}
}

```

2. 增、删、改、查数据块

BlockManager 最重要的功能之一就是维护 Namenode 内存中的数据块信息,BlockManager 中存储的数据块信息包含两个部分。

- 数据块与存储这个数据块的数据节点存储的对应关系,这部分信息保存在数据块对应的 BlockInfo 对象的 triplets[]数组中(请参考 BlockInfo 类小节),Namenode 内存中的所有 BlockInfo 对象则保存在 BlockManager.blocksMap 字段中。
- 数据节点存储与这个数据节点存储上保存的所有数据块的对应关系,这部分信息保存在 DatanodeStorageInfo.blockList 字段中,blockList 是 BlockInfo 类型的,利用 BlockInfo.triplets[] 字段的双向链表结构(请参考 BlockInfo 类小节),DatanodeStorageInfo 可以通过 blockList 字段保存这个数据块存储上所有数据块对应的 BlockInfo 对象。

BlockManager 管理 Namenode 中数据块的信息基本上都是通过对这两部分内容进行修改实现的,本节就介绍 BlockManager 中数据块的增、删、改、查相关方法。

(1) 添加数据块

当客户端向 HDFS 写入新文件时,如果写满了一个数据块,客户端会调用 ClientProtocol.addBlock()方法向 Namenode 申请一个新的数据块。这个请求到达 Namenode 后会由 FSNamesystem.getAdditionalBlock()方法响应,对 FSNamesystem.getAdditionalBlock()方法的分析请参考 ClientProtocol 实现中的创建新的数据块小节。

getAdditionalBlock()方法首先会检查文件系统状态,然后为新添加的数据块选择存放副本的 Datanode,最后构造 Block 对象并调用 FSDirectory.addBlock()方法将 Block 对象加入文件对应的 INode 对象中。再来看 FSDirectory.addBlock()方法,它首先会构造 Block 对应的 BlockInfo 对象,然后调用 BlockManager.addBlockCollection()方法将这个 BlockInfo 对象加入 BlockManager.blocksMap 字段中存储,最后 addBlock()会调用 INodeFile.addBlock()方法将 BlockInfo 对象添加到 INodeFile 对象的 blocks 字段中保存。对 FSDirectory.addBlock()方法的分析请参考文件系统目录树中的 FSDirectory 类小节。

我们这里关注一下 BlockManager.addBlockCollection()方法的实现,这个方法用于在 HDFS 第二关系中添加一个数据块的信息,也就是在 BlockManager.blocksMap 字段中添加一个 BlockInfo 对象。addBlockCollection()方法的实现很简单,就是在 blocksMap 对象上调用 addBlockCollection()方法。

```
public BlockInfo addBlockCollection(BlockInfo block, BlockCollection bc) {  
    // 将BlockInfo对象添加到BlockManager.blocksMap中  
    return blocksMap.addBlockCollection(block, bc);  
}
```

到这里，客户端就将一个数据块完整地添加到 Namenode 的内存中了，之后客户端就可以通过数据流管道向 Datanode 写入这个数据块了。那么当数据流管道中的 Datanode 上写入了一个数据块的副本后，副本的信息是如何添加到 Namenode 中的呢？我们继续看下面的分析。

(2) 添加副本

当 Datanode 上写入了一个新的数据块副本或者完成了一次数据块副本复制操作后，会通过 `DatanodeProtocol.blockReport()` 或者 `DatanodeProtocol.blockReceivedAndDeleted()` 方法向 Namenode 汇报该 Datanode 上添加了一个新的数据块副本。这两个接口最终都会通过调用 `BlockManager.addStoredBlock()` 方法更新 `BlockManager.blocksMap` 中的数据块副本与 Datanode 的对应信息。本节就分析 `addStoredBlock()` 的具体实现。

图 3-39 给出了 `addStoredBlock()` 方法的流程图。`addStoredBlock()` 方法不仅仅完成了将数据块副本信息添加到内存的操作，同时还实现了更新数据块状态，以及更新 `neededReplications` 队列、`excessReplicateMap` 队列、`corruptReplicas` 队列等功能。所以这个方法的代码比较长，我们一段段分析。



图 3-39 `addStoredBlock()` 方法流程图

`addStoredBlock()` 方法首先确认当前副本是否属于 Namenode 内存中的一个 HDFS 文件，如果不属于则直接返回。然后 `addStoredBlock()` 会调用 `storageInfo.addBlock()` 在数据块与数据节点存储的映射中添加当前数据节点存储的信息（在 `BlockInfo` 的 `triplets[]` 数组中添加当前

DatanodeStorageInfo 的信息），并在当前数据节点存储对象上添加这个数据块的信息（在 DatanodeStorageInfo 的 blockList 链表中添加当前副本对应的 BlockInfo 对象），storageInfo.addBlock() 方法的实现请参考数据节点管理中的 DatanodeStorageInfo 小节的介绍。

```
private Block addStoredBlock(final BlockInfo block,
                             DatanodeStorageInfo storageInfo,
                             DatanodeDescriptor delNodeHint,
                             boolean logEveryBlock)
    throws IOException {

    // ...
    if (storedBlock == null || storedBlock.getBlockCollection() == null) {
        // 如果当前 block 不属于任何 inode，则直接返回，不进行任何操作
        return block;
    }
    BlockCollection bc = storedBlock.getBlockCollection();

    // 在 block -> datanode 映射中添加当前 datanode
    boolean added = storageInfo.addBlock(storedBlock);
    // ...
}
```

如果新添加的副本对应数据块的状态为 COMMITTED（请参考 Block 类状态小节），也就是客户端已经提交了当前数据块（客户端已经将数据块的所有数据写入数据流管道中），addStoredBlock()方法会调用 completeBlock()方法将 Namenode 中保存的当前数据块的状态由构建状态转换为正常状态，也就是将 Namenode 文件系统目录树的 INode 对象以及 BlockManager.blocksMap 字段中保存的 BlockInfoUnderConstruction 引用更新为正常的 BlockInfo 引用。如果数据块不处于 COMMITTED 状态，则跳出继续执行。completeBlock()方法只有在当前数据块处于提交状态，并且数据块的副本数量满足最小副本要求时才会被调用。

```
if(storedBlock.getBlockUCState() == BlockUCState.COMMITTED &&
    numLiveReplicas >= minReplication) {
    storedBlock = completeBlock((MutableBlockCollection)bc, storedBlock, false);
} else if (storedBlock.isComplete()) {
    namesystem.incrementSafeBlockCount(numCurrentReplica);
}

if (bc instanceof MutableBlockCollection) {
    return storedBlock;
}
```

接下来 addStoredBlock()方法会调用 isNeededReplication()判断当前数据块的副本数量是否满足期望，也就是用户配置的副本系数。如果已经满足了期望，则该数据块没有必要进行复制操作，从 BlockManager.neededReplications 队列中删除这个数据块；如果不满足期望，则调用 updateNeededReplications()判断数据块需要复制的次数，然后更新 BlockManager.neededReplications 队列。

```
short fileReplication = bc.getBlockReplication();
if (!isNeededReplication(storedBlock, fileReplication, numCurrentReplica)) {
    neededReplications.remove(storedBlock, numCurrentReplica,
        num.decommissionedReplicas(), fileReplication);
} else {
    updateNeededReplications(storedBlock, curReplicaDelta, 0);
}
if (numCurrentReplica > fileReplication) {
    processOverReplicatedBlock(storedBlock, fileReplication, node, delNodeHint);
}
```

除此之外，`addStoredBlock()`方法还会判断数据块当前的副本数量是否已经超出了期望，如果超出了则存在多余副本，调用 `processOverReplicatedBlock()`方法将超出的副本放入 `BlockManager.excessReplicateMap` 队列中。对 `processOverReplicatedBlock()`方法的分析请参考后续的删除副本小节。

```
if (numCurrentReplica > fileReplication) {
    processOverReplicatedBlock(storedBlock, fileReplication, node, delNodeHint);
}
```

接下来处理数据块的损坏副本，`Namenode` 会保证只有在数据块拥有足够的副本数量时才处理这个数据块的损坏副本。所以添加了新的副本后，如果该数据块的有效副本数量已经超过了期望，`addStoredBlock()`方法就会调用 `invalidateCorruptReplicas()`将该数据块所有的损坏副本从 `Datanode` 上删除，并且从 `BlockManager.blocksMap` 字段中删除。对 `invalidateCorruptReplicas()`方法的分析请参考后续的删除副本小节。

```
int corruptReplicasCount = corruptReplicas.numCorruptReplicas(storedBlock);
int numCorruptNodes = num.corruptReplicas();
if (numCorruptNodes != corruptReplicasCount) {
}
if ((corruptReplicasCount > 0) && (numLiveReplicas >= fileReplication))
    invalidateCorruptReplicas(storedBlock);
return storedBlock;
```

至此，一个完整的添加副本流程就结束了。可以看到，向 `Namenode` 中添加一个新的副本后，会引起该副本对应数据块状态的改变，这里读者需要特别注意。

(3) 删除数据块

当客户端删除一个 HDFS 文件时，客户端会调用 RPC 接口 `ClientProtocol.delete()`删除 HDFS 文件或者目录，并且删除文件拥有的所有数据块，以及这个数据块在 `Datanode` 上的所有副本，这个请求会在 `FSNamesystem.delete()` -> `deleteInt()` -> `deleteInternal()`方法中响应。

`deleteInternal()`方法首先会调用 `FSDirectory.delete()`方法将文件对应的 `Inode` 对象从文件系统目录树中删除，然后将这个 `Inode` 下保存的所有数据块收集到 `collectedBlocks` 集合中，如果被删除的这个 `Inode` 是一个文件夹，则将这个文件夹下级联删除的所有 `Inode` 放入 `removedINodes` 集合中，最后更新父节点的修改时间。然后 `deleteInternal()`方法会调用 `removePathAndBlocks()`方法将 `removedINodes` 集合中的所有 `Inode` 对象从文件系统目录树根

节点的 `inodeMap` 中删除, 并从租约管理器中删除这些 `Inode` 文件的租约。最后 `deleteInternal()` 方法会调用 `removeBlocks()` 方法删除 `Inode` 中保存的所有数据块, 也就是 `collectedBlocks` 集合中收集的所有数据块。

```
private boolean deleteInternal(String src, boolean recursive,
    boolean enforcePermission, boolean logRetryCache)
    throws AccessControlException, SafeModeException, UnresolvedLinkException,
        IOException {
    // ...
    try {
        // 检查是否有写权限, 以及当前节点是否处于安全模式中
        checkOperation(OperationCategory.WRITE);
        checkNameNodeSafeMode("Cannot delete " + src);
        src = resolvePath(src, pathComponents);

        // 调用 FSDirectory 将 Inode 从文件系统目录树中移出
        // 并将该 Inode 对应的所有数据块放入 collectedBlocks 集合中保存
        // 如果当前 Inode 是目录, 则将该目录的所有子节点 Inode 放入 removedINodes 中
        long filesRemoved = dir.delete(src, collectedBlocks, removedINodes,
            mtime);
        if (filesRemoved < 0) {
            return false;
        }
        // 将删除操作记录在 editlog 中
        getEditLog().logDelete(src, mtime, logRetryCache);
        // 将 removeINodes 集合中的所有 Inode 从 HDFS 根节点的 inodeMap 中删除
        // 并在 LeaseManager 中移出 removeINodes 中所有 Inode 的租约
        removePathAndBlocks(src, null, removedINodes, true);
        ret = true;
    } finally {
        writeUnlock();
    }
    getEditLog().logSync();
    // 调用 removeBlocks() 删除 collectedBlocks 集合中的所有数据块
    removeBlocks(collectedBlocks);
    collectedBlocks.clear();
    return ret;
}
```

`removeBlocks()` 方法会遍历 `collectedBlocks` 中的所有数据块, 然后调用 `BlockManager.removeBlock()` 方法将该数据块从 Namenode 中完全删除, 包括 `blocksMap`、`postponedMisreplicatedBlocksCount`、`pendingReplications`、`neededReplications`、`corruptReplicas` 等 `BlockManager` 操作队列中保存的这个数据块的信息, 以及这个数据块的副本信息。之后 `removeBlocks()` 会调用 `addToInvalidates()` 方法将该数据块的所有副本从 Datanode 上删除。

```
public void removeBlock(Block block) {
    assert namesystem.hasWriteLock();
    block.setNumBytes(BlockCommand.NO_ACK);
```

```
// 将该数据块的所有副本加入 invalidateBlocks 队列中，从保存它们的数据节点上删除
addToInvalidates(block);
corruptReplicas.removeFromCorruptReplicasMap(block);
// 从 blocksMap 中删除
blocksMap.removeBlock(block);
// 从 pendingReplications、neededReplications 队列中删除
pendingReplications.remove(block);
neededReplications.remove(block, UnderReplicatedBlocks.LEVEL);
// 从 postponedMisreplicatedBlocksCount 队列中删除
if (postponedMisreplicatedBlocks.remove(block)) {
    postponedMisreplicatedBlocksCount.decrementAndGet();
}
}
```

`addToInvalidates()`方法的实现也非常简单，它会遍历所有保存这个数据块副本的数据节点，然后将这个数据节点保存的副本加入 `invalidateBlocks` 队列中。`invalidateBlocks` 队列中的副本会由 `ReplicationMonitor` 线程处理，这部分内容我们在 `ReplicationMonitor` 小节中已经介绍过了，请读者参考。

```
private void addToInvalidates(Block b) {
    if (!namesystem.isPopulatingReplQueues()) {
        return;
    }
    StringBuilder datanodes = new StringBuilder();
    // 遍历保存这个数据块副本的所有数据节点，从这些节点上删除这个副本
    for (DatanodeStorageInfo storage : blocksMap.getStorages(b, State.NORMAL)) {
        final DatanodeDescriptor node = storage.getDatanodeDescriptor();
        invalidateBlocks.add(b, node, false);
        datanodes.append(node).append(" ");
    }
}
```

至此，一个完整的数据块删除流程就结束了。

(4) 删除副本

上一节我们介绍了 `BlockManager` 中删除数据块的流程，由于一个数据块会在多个 `Datanode` 上存储数据块副本，所以本节介绍数据块副本的删除操作。图 3-40 给出了 HDFS 数据块副本删除流程图。

数据块副本在以下三种情况下会被删除。

- 数据块所属的 HDFS 文件被删除，这个文件对应的所有数据块以及副本都将被删除。用户通过调用 `ClientProtocol.delete()` 方法请求删除 HDFS 的一个文件，这个删除请求会由 `FSNamesystem.delete()` 方法响应，`delete()` 方法会将 HDFS 文件从文件系统目录树中删除，同时调用 `BlockManager.removeBlocks()` 方法将文件拥有的所有数据块以及数据块的所有副本删除。`removeBlocks()` 方法首先将数据块的所有副本加入 `invalidateBlocks` 队列中执行删除操作，然后再从 `BlockManager.blocksMap`、

pendingReplications、neededReplications 以及 postponedMisreplicatedBlocks 中删除所有的数据块信息。invalidateBlocks 队列的处理我们在后面统一介绍。

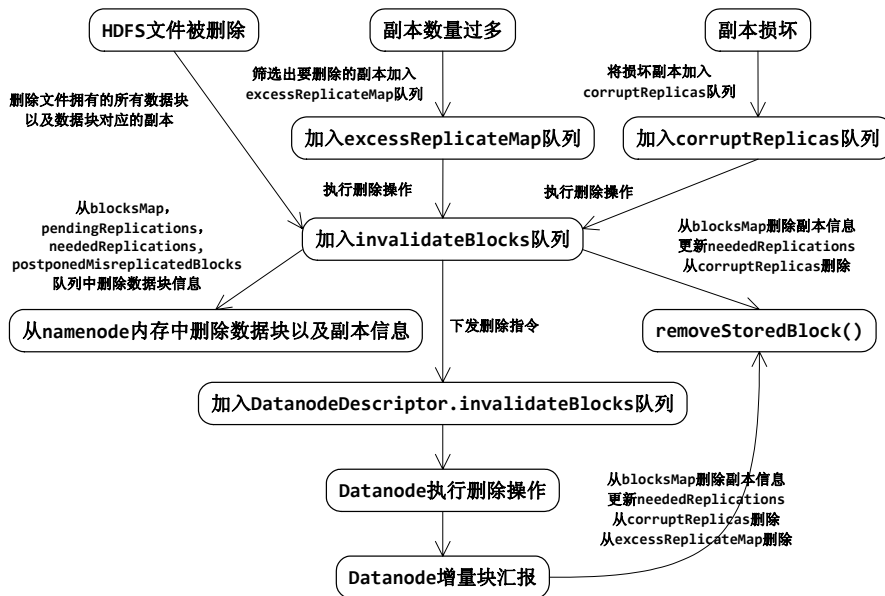


图 3-40 数据块副本删除流程图

- 数据块的副本数量多于配置的副本系数时，多出的副本会被删除。BlockManager 发现数据块的副本数量过多时，就会调用 processOverReplicatedBlock() 方法删除多余的副本。processOverReplicatedBlock() 方法会调用 chooseExcessReplicates() 从数据块的所有副本中筛选出多余的副本执行删除操作，之后 processOverReplicatedBlock() 方法会将这些多余的副本加入 excessReplicateMap 中暂存，最后再将这些副本加入 invalidateBlocks 队列中执行删除操作。当 Datanode 成功地执行了删除操作后，会通过增量块汇报通知 BlockManager，BlockManager 会调用 removeStoredBlock() 方法从内存中删除副本信息，并将 excessReplicateMap 中暂存的副本信息删除。
- 副本被 Namenode 标识为损坏副本，需要删除。客户端或者数据节点发现损坏的副本后通知 Namenode，Namenode 会调用 BlockManager.markBlockAsCorrupt() 方法处理损坏的副本。markBlockAsCorrupt() 方法会直接将损坏的副本加入 corruptReplicas 队列中暂存，然后再将这些副本加入 invalidateBlocks 队列中执行删除操作。接下来 markBlockAsCorrupt() 方法会直接调用 removeStoredBlock() 从 BlockManager 内存中删除副本的信息。需要注意对于损坏副本的处理不同于多余副本。markBlockAsCorrupt() 并不等待数据节点通知 BlockManager 数据块副本已经删除，而是直接调用 removeStoredBlock() 从内存中删除副本信息。

下面我们再学习 invalidateBlocks 队列的处理。在 ReplicationMonitor 小节中我们已经介绍了，ReplicationMonitor 线程会定期调用 computeInvalidateWork() 方法从 invalidateBlocks 队列

中筛选出执行删除操作的副本，然后 `computeInvalidateWork()` 会将这些待删除副本加入 `DatanodeDescriptor.invalidateBlocks` 队列中产生删除指令，最后通过心跳响应将指令带回 `Datanode` 节点。当 `Datanode` 成功地执行了副本删除操作后，`Datanode` 会通过增量块汇报接口通知 `BlockManager`，这时 `BlockManager` 会调用 `removeStoredBlock()` 方法将副本信息从 `blocksMap`、`neededReplications`、`corruptReplicas` 以及 `excessReplicateMap` 中删除。至此，一个完整的数据块副本删除操作就完成了。

下面我们依次分析处理这三种情况的代码实现。

HDFS 文件被删除

当 HDFS 文件被删除时，这个文件所属的所有数据块也会被删除，同时这些数据块在 `Datanode` 上存储的所有副本也会被删除。HDFS 文件被删除的情况，我们在删除数据块小节中已经介绍了，这里就不再重复介绍了。

多余副本的删除

HDFS 管理员可以对 HDFS 文件设置副本系数，即一个数据块在 HDFS 中可以有多少个副本，如果指定数据块在所有 `Datanode` 上存储的副本数量超过了副本系数，HDFS 会将多余的副本删除。多余副本的扫描操作是由 `processOverReplicatedBlock()` 方法实现的，图 3-41 给出了触发副本扫描操作的几种情况。



图 3-41 processOverReplicatedBlock()调用关系

- `setReplication()`——管理员更改了当前 HDFS 文件的副本系数，如果新设置的副本系数小于原来的副本系数，那么 HDFS 集群中一定存在多余的副本，这时调用 `processOverReplicatedBlock()` 处理多余的副本。
- `addStoredBlock()`——当 `Datanode` 通过 `DatanodeProtocol.blockReceivedAndDeleted()` 汇报 `Datanode` 新添加了一个数据块副本时，可能会出现数据块的副本数量超过了副本系数的情况，此时就需要调用 `processOverReplicatedBlock()` 处理多余的副本。
- `processOverReplicatedBlocksOnReCommission()`——我们知道，当撤销（`decommission`）一个 `Datanode` 时，会将该节点上所保存的副本进行复制操作，以保证这些数据块满足副本系数，完成这个操作后 `Datanode` 才能正常退役（请参考数据节点管理小节）。如果我们将一个撤销的节点重新上架，这个节点上保存的副本会重新回到 HDFS 集群中，也就可能产生多余的副本，这时就需要调用 `processOverReplicatedBlock()` 处理多余的副本。
- `rescanPostponedMisreplicatedBlocks()`——当 `Namenode` 发生错误，进行 `Active` 和 `Standby` 切换时，HDFS 会延迟副本的删除操作，并将这些要删除的副本放入

postponedMisreplicatedBlocks 队列中缓存。当 Datanode 重新向当前的 ActiveNamenode 发送心跳时，Namenode 会调用 rescanPostponedMisreplicatedBlocks() 方法重新扫描 postponedMisreplicatedBlocks 队列中保存的待删除副本，这里就包括判断该副本在当前情况下是否还是多余的副本，如果存在多余的副本，则调用 processOverReplicatedBlock() 方法处理。

- processMisReplicatesAsync()——当 Namenode 进入 Active 状态时，会调用这个方法对命名空间中的所有数据块重新扫描，并将扫描出的无效数据块加入 invalidateBlocks 队列中，有 stale 副本的数据块放入 postponedMisreplicatedBlocks 队列中，超过副本系数的数据块调用 processOverReplicatedBlock() 方法处理。
- checkReplication()——当客户端完成一个文件的写操作后，会检查这个文件拥有的所有数据块是否满足副本系数，如果超出副本系数，则调用 processOverReplicatedBlock() 处理。

下面我们来学习 processOverReplicatedBlock() 方法的实现。processOverReplicatedBlock() 方法首先筛选出可以进行副本删除操作的 Datanode，它遍历所有保存这个数据块的 Datanode，剔除已经在 excessReplicateMap 中的节点、正在撤销的节点、已经撤销的节点，以及副本已经损坏的节点，剩余的所有满足条件的 Datanode 放入 nonExcess 队列中保存。完成了 Datanode 的筛选操作后，processOverReplicatedBlock() 方法会调用 chooseExcessReplicates() 方法在 nonExcess 队列中选出执行删除操作的节点并执行删除操作。

```
private void processOverReplicatedBlock(final Block block,
    final short replication, final DatanodeDescriptor addedNode,
    DatanodeDescriptor delNodeHint) {
    assert namesystem.hasWriteLock();
    if (addedNode == delNodeHint) {
        delNodeHint = null;
    }
    Collection<DatanodeStorageInfo> nonExcess = new ArrayList<DatanodeStorageInfo>();
    Collection<DatanodeDescriptor> corruptNodes = corruptReplicas
        .getNodes(block);
    for(DatanodeStorageInfo storage : blocksMap.getStorages(block, State.NORMAL)) {
        final DatanodeDescriptor cur = storage.getDatanodeDescriptor();
        if (storage.areBlockContentsStale()) {
            postponeBlock(block);
            return;
        }
    }
    LightweightLinkedSet<Block> excessBlocks = excessReplicateMap.get(cur
        .getDatanodeUuid());
    if (excessBlocks == null || !excessBlocks.contains(block)) {
        if (!cur.isDecommissionInProgress() && !cur.isDecommissioned()) {
            if (corruptNodes == null || !corruptNodes.contains(cur)) {
                nonExcess.add(storage);
            }
        }
    }
}
```

```

    }
    chooseExcessReplicates(nonExcess, block, replication,
        addedNode, delNodeHint, blockplacement);
}

```

`chooseExcessReplicates()`是一个比较大的方法，它首先从 `processOverReplicatedBlock()`筛选出的 `nonExcess` 队列中选出满足条件的 `Datanode`，然后调用 `addToExcessReplicate()`方法将这个节点上的多余副本加入 `excessReplicateMap` 中，最后调用 `addToInvalidates()`方法将多余的副本加入 `invalidateBlocks` 队列中执行删除操作。`excessReplicateMap` 是用来跟踪多余副本删除情况的队列。`processOverReplicatedBlock()`选择执行删除操作的 `Datanode` 的原则是，尽量保证在删除操作后副本能均匀地分布在不同的机架上，并且尽量从空间较少的节点上删除冗余副本，选择的算法可以抽象为以下两步。

- 首先找出保存这个数据块副本最多的机架，然后找到这个机架上磁盘空间最小的节点，选择这个节点删除副本。这样就保证了副本分布在足够多的机架上。
- 如果不存在这样的副本，则选择一个空间最小的节点。

```

private void chooseExcessReplicates(final Collection<DatanodeStorageInfo> nonExcess,
    Block b, short replication,
    DatanodeDescriptor addedNode,
    DatanodeDescriptor delNodeHint,
    BlockPlacementPolicy replicator) {
    assert namesystem.hasWriteLock();
    BlockCollection bc = getBlockCollection(b);
    final BlockStoragePolicy storagePolicy=storagePolicySuite.getPolicy(bc.getStoragePolicyID());
    final List<StorageType> excessTypes = storagePolicy.chooseExcess(
        replication, DatanodeStorageInfo.toStorageTypes(nonExcess));

    final Map<String, List<DatanodeStorageInfo>> rackMap
        = new HashMap<String, List<DatanodeStorageInfo>>();
    final List<DatanodeStorageInfo> moreThanOne = new ArrayList<DatanodeStorageInfo>();
    final List<DatanodeStorageInfo> exactlyOne = new ArrayList<DatanodeStorageInfo>();

    boolean firstOne = true;
    final DatanodeStorageInfo delNodeHintStorage
        = DatanodeStorageInfo.getDatanodeStorageInfo(nonExcess, delNodeHint);
    final DatanodeStorageInfo addedNodeStorage
        = DatanodeStorageInfo.getDatanodeStorageInfo(nonExcess, addedNode);
    while (nonExcess.size() - replication > 0) {
        final DatanodeStorageInfo cur;
        if (useDelHint(firstOne, delNodeHintStorage, addedNodeStorage,
            moreThanOne, excessTypes)) {
            cur = delNodeHintStorage;
        } else { // regular excessive replica removal
            cur = replicator.chooseReplicaToDelete(bc, b, replication,
                moreThanOne, exactlyOne, excessTypes);
        }
    }
}

```



```

firstOne = false;

replicator.adjustSetsWithChosenReplica(rackMap, moreThanOne,
    exactlyOne, cur);

nonExcess.remove(cur);
// 加入 excessReplicateMap 队列
addToExcessReplicate(cur.getDatanodeDescriptor(), b);
// 加入 invalidateBlocks 队列
addToInvalidates(b, cur.getDatanodeDescriptor());
}
}

```

chooseExcessReplicates() 方法调用 addToInvalidates() 方法将待删除的副本加入 invalidateBlocks 队列后, BlockManager 的 ReplicationMonitor 线程会处理 invalidateBlocks 队列中的所有待删除的副本。

```

void addToInvalidates(final Block block, final DatanodeInfo datanode) {
    if (!namesystem.isPopulatingReplQueues()) {
        return;
    }
    invalidateBlocks.add(block, datanode, true);
}

```

当 Datanode 成功地删除了冗余的数据块副本后, 会通过增量块汇报接口通知 BlockManager, BlockManager 会调用 BlockManager.removeStoredBlock() 方法删除 excessReplicateMap 中保存的多余的数据块副本信息。下面我们再看一下 removeStoredBlock() 方法的实现。removeStoredBlock() 的作用和 addStoredBlock() 正好相反, 它不仅从 blocksMap 中删除了这个副本, 同时还更新了 Namenode 上保存这个副本的其他数据结构。我们知道, invalidateBlocks 队列中的副本主要来自于 excessBlocks 和 corruptReplicas, 所以当我们从一个数据节点上删除这个副本后, 该副本只有可能是冗余副本或者损坏副本, 完成了删除操作之后即可检查这两个队列, 看删除副本是否来自于这两个队列, 如果来自于这两个队列, 则从中删除。

```

public void removeStoredBlock(Block block, DatanodeDescriptor node) {

    assert (namesystem.hasWriteLock());
    {
        // 从 blocksMap 中删除这个数据节点上的副本
        if (!blocksMap.removeNode(block, node)) {
            return;
        }

        // 这次删除操作有可能是数据节点发生错误, 但是该数据块依然有效
        // 在这种情况下数据块的副本数量可能不足, 更新 neededReplications 队列
        BlockCollection bc = blocksMap.getBlockCollection(block);
        if (bc != null) {
            namesystem.decrementSafeBlockCount(block);
        }
    }
}

```

```
        updateNeededReplications(block, -1, 0);
    }

    // 已经从该数据节点上删除了副本, 更新 excessReplicateMap 队列
    LightweightLinkedSet<Block> excessBlocks = excessReplicateMap.get(node
        .getDatanodeUuid());
    if (excessBlocks != null) {
        if (excessBlocks.remove(block)) {
            excessBlocksCount.decrementAndGet();
            if (excessBlocks.size() == 0) {
                excessReplicateMap.remove(node.getDatanodeUuid());
            }
        }
    }
}

// 已经从该数据节点上删除了副本, 更新 corruptReplicas 队列
corruptReplicas.removeFromCorruptReplicasMap(block, node);
}
}
```

损坏副本的删除

客户端读文件以及数据节点的数据块扫描器都可能发现损坏的数据块副本, 客户端会通过 `ClientProtocol.reportBadBlocks()` 方法向 `Namenode` 汇报损坏的数据块副本, 数据节点会通过 `DatanodeProtocol.reportBadBlocks()` 方法向 `Namenode` 汇报损坏的数据块副本。还有一种情况是, 在数据节点的块汇报以及增量块汇报操作时, `Namenode` 会将汇报的数据块副本信息与当前 `Namenode` 内存中的数据块信息进行对比, 然后计算出损坏的数据块副本 (请参考块汇报小节)。在上述几种情况下, `Namenode` 会调用 `markBlockAsCorrupt()` 方法处理损坏的副本。

`markBlockAsCorrupt()` 方法会将损坏的副本加入 `corruptReplicas` 队列中, 然后判断该副本对应的数据块是否有足够的副本数量, 如果数据块已经有足够的备份数量, 则将调用 `invalidateBlock()` 方法直接将损坏的副本加入 `invalidateBlocks` 队列中进行删除操作。如果副本系数不足, 则更新 `neededReplications` 队列, 复制该数据块。

```
private void markBlockAsCorrupt(BlockToMarkCorrupt b,
    DatanodeStorageInfo storageInfo,
    DatanodeDescriptor node) throws IOException {

    BlockCollection bc = b.corrupted.getBlockCollection();
    // 如果当前数据块副本不属于任何文件, 则直接删除
    // 例如, 该数据块副本对应的 HDFS 文件已经被删除了, 但数据节点上的副本还未删除
    if (bc == null) {
        addToInvalidates(b.corrupted, node);
        return;
    }

    if (storageInfo != null) {
        storageInfo.addBlock(b.stored);
    }
}
```

```

    }

    // 在 corruptReplicas 队列中添加这个数据块副本
    corruptReplicas.addToCorruptReplicasMap(b.corrupted, node, b.reason,
        b.reasonCode);

    NumberReplicas numberOfReplicas = countNodes(b.stored);
    boolean hasEnoughLiveReplicas = numberOfReplicas.liveReplicas() >= bc
        .getBlockReplication();
    boolean minReplicationSatisfied =
        numberOfReplicas.liveReplicas() >= minReplication;
    boolean hasMoreCorruptReplicas = minReplicationSatisfied &&
        (numberOfReplicas.liveReplicas() + numberOfReplicas.corruptReplicas()) >
        bc.getBlockReplication();
    boolean corruptedDuringWrite = minReplicationSatisfied &&
        (b.stored.getGenerationStamp() > b.corrupted.getGenerationStamp());
    // 如果数据块有足够的副本数量, 则直接删除这个副本
    if (hasEnoughLiveReplicas || hasMoreCorruptReplicas
        || corruptedDuringWrite) {
        invalidateBlock(b, node);
    } else if (namesystem.isPopulatingReplQueues()) {
        // 如果数据块的副本系数不足, 则复制这个数据块
        updateNeededReplications(b.stored, -1, 0);
    }
}

```

使用 `invalidateBlock()` 方法删除副本的操作主要包括两个部分: ① 调用 `addToInvalidates()` 方法将数据块加入 `invalidateBlocks` 队列中, 之后 `BlockManager` 会将这个副本的删除指令通过心跳响应发送给 `Datanode`; ② 调用 `removeStoredBlock()` 从 `BlockManager.blocksMap` 中移除这个数据节点上的副本信息, 同时更新 `excessReplicateMap`、`corruptReplicas` 以及 `neededReplications` 队列。对 `removeStoredBlock()` 方法的分析请参考上一节内容。

```

private boolean invalidateBlock(BlockToMarkCorrupt b, DatanodeInfo dn
    ) throws IOException {
    DatanodeDescriptor node = getDatanodeManager().getDatanode(dn);
    if (node == null) {
        throw new IOException("Cannot invalidate " + b
            + " because datanode " + dn + " does not exist.");
    }

    // 检查当前数据块有多少个副本
    NumberReplicas nr = countNodes(b.stored);
    if (nr.replicasOnStaleNodes() > 0) {
        // 如果有副本在 stale 状态的节点上, 那么对于这个副本的删除操作延迟
        // 直到 stale 状态的副本恢复正常
        postponeBlock(b.corrupted);
        return false;
    } else if (nr.liveReplicas() >= 1) {

```

```
// 如果当前数据块至少有一个有效副本，则进行删除操作
addToInvalidates(b.corrupted, dn);
removeStoredBlock(b.stored, node);
return true;
} else {
    return false;
}
}
```

这里还有一个很有意思的地方，就是对于多余副本的处理是在 `processOverReplicatedBlocks()` 中进行的，这个方法会将多余的副本加入 `excessReplicateMap` 队列中记录，然后加入 `invalidateBlocks` 队列中触发 `Datanode` 的删除操作，当 `Datanode` 完成删除操作后，会调用 `DatanodeProtocol.blockReceivedAndDeleted()` 方法向 `Namenode` 进行增量汇报，最后 `BlockManager` 会调用 `removeStoredBlock()` 方法删除 `Namenode` 内存中保存的副本信息，并从 `excessReplicateMap` 队列中删除这个副本的信息。不同于对于多余副本的处理，处理损坏副本的 `markBlockAsCorrupt()` 方法会先调用 `removeStoredBlock()` 从 `Namenode` 内存中删除该副本，包括从 `corruptReplicas` 队列中删除副本信息，然后才将这个副本加入 `invalidateBlocks` 队列中触发 `Datanode` 的删除操作。当 `Datanode` 完成删除操作并通知 `Namenode` 后，`Namenode` 调用 `removeStoredBlock()` 方法时会发现 `Namenode` 内存中的副本信息已经删除，无须再进行删除操作了。

为什么对于损坏的数据块需要先调用 `removeStoredBlock()` 从 `Namenode` 内存中删除副本的信息呢？这是为了防止 `Namenode` 向 `Datanode` 下发的副本删除指令丢失，如果 `Datanode` 没有删除损坏的副本，那么在下次块汇报时 `Datanode` 会再次上报这个损坏的副本信息，这时由于 `Namenode` 内存中并没有保存这个副本的信息，`Namenode` 会再次将这个副本标识为损坏的副本并执行删除操作。

(5) 数据块的复制

数据块复制操作是 HDFS 保证数据块冗余存储的一个重要特性，也体现了 HDFS 故障检测和自动恢复的特性。在 `Namenode` 的许多流程中都包括了数据块复制操作，例如撤销节点、处理损坏的数据块副本等操作。数据块复制流程是 HDFS 中最重要的流程之一，图 3-42 给出了 HDFS 数据块复制操作的流程图。

`Namenode` 会在以下三种情况下将一个数据块副本加入 `neededReplications` 队列中以执行数据块副本复制流程。① 客户端完成了一个文件的写操作，`Namenode` 会检查这个文件包含的所有数据块是否有足够的副本数量，如果不足则加入 `neededReplications` 队列中。② 当 `Namenode` 执行一个 `Datanode` 的撤销操作时，会将这个 `Datanode` 上保存的所有副本进行复制，也就是将这些副本加入 `neededReplications` 队列中。③ `pendingReplications` 队列中保存的数据块复制任务超时，会将这些任务重新加入 `neededReplications` 队列中。

当待复制数据块加入 `neededReplications` 队列后，会由 `ReplicationMonitor` 线程处理（请参考 `BlockManger` 类的 `ReplicationMonitor` 小节），`ReplicationMonitor` 线程会调用 `computeDatanodeWork()` 方法定期从 `neededReplications` 队列中筛选出数据块，然后为这些数据块选择

复制的源 Datanode 和目标 Datanode，再将数据块加入源 Datanode 对应的 DatanodeDescriptor 的 replicateBlocks 队列，生成复制数据块的指令通过心跳响应将指令带到源 Datanode。完成名字节点指令的生成后，computeDatanodeWork()方法会将待复制数据块从 neededReplications 队列中删除，然后加入 pendingReplications 队列。

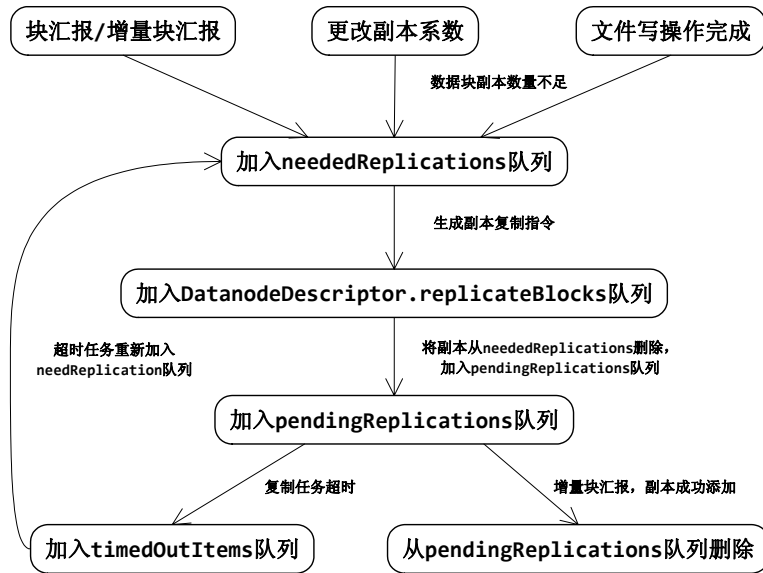


图 3-42 数据块复制流程图

pendingReplications 队列用于存放已经生成复制请求的数据块，PendingReplicationMonitor 线程（请参考 BlockManager 数据结构的 PendingReplicationBlocks 类小节）会定期调用 pendingReplicationCheck()方法扫描 pendingReplications 中保存的所有数据块，如果发现有复制请求超时，则将超时的请求放入 PendingReplicationBlocks.timeOutItems 队列中，然后由 ReplicationMonitor 调用 processPendingReplications()方法将 timeOutItems 队列中的复制请求重新加回 neededReplications 队列。另一种情况是，复制请求成功执行，Datanode 在写入了数据块副本后通过增量汇报接口通知 Namenode，这时 BlockManager 会调用 BlockManager.add()方法将这个副本信息添加到 BlockManager 中，并且从 pendingReplications 中删除这个复制请求。至此，一个闭环的数据块复制操作就完成了。

ReplicationMonitor 和 PendingReplicationMonitor 线程我们已经在前面的小节中介绍过了，本节重点介绍数据块复制流程中添加数据块复制操作以及更新数据块复制操作的代码实现。

添加数据块复制操作

添加一个数据块复制操作是通过调用 neededReplicationsUnderReplicatedBlocks.add()方法进行的，add()方法会将待复制的数据块加入 neededReplications 队列中。如图 3-43 所示，添加数据块复制操作是在如下几种情况下进行的。

- 当 Namenode 处理块汇报中的副本时，例如添加副本操作（`addStoredBlock()`）、删除副本操作（`removeStoredBlock()`）、标识损坏的副本操作（`markBlockAsCorrupt()`），会判断汇报副本对应的数据块是否有足够的副本数量。如果数据块的副本数量不足，这些方法会调用 `BlockManager.updateNeededReplications()` 方法将这个数据块加入 `neededReplications` 队列中以执行数据块复制操作。
- 当客户端完成了对一个 HDFS 文件的写操作时，会调用 `finalizeINodeFileUnderConstruction()` 关闭这个文件，这时 `finalizeINodeFileUnderConstruction()` 会遍历文件中所有的数据块并调用 `BlockManager.checkReplication()` 方法判断这些数据块是否有足够的副本数量，如果副本数量不足则加入 `neededReplications` 队列中。
- 当用户更改了数据块的副本系数时，会调用 `BlockManager.setReplication()` 方法执行更新操作，`setReplication()` 会调用 `BlockManager.updateNeededReplications()` 方法更新 `neededReplications` 队列，在副本系数不足的时候将数据块添加至 `neededReplications` 队列中。

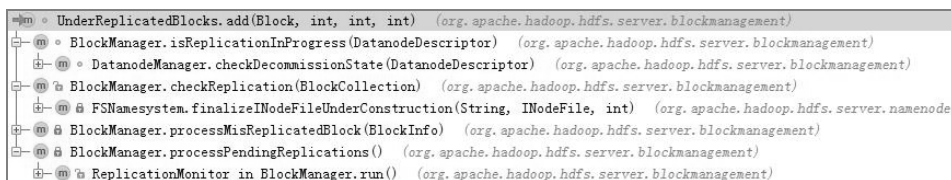


图 3-43 UnderReplicatedBlocks.add()调用关系

我们知道 `UnderReplicatedBlocks` 底层是一个优先级队列，优先级的确定以及实现请参考 `UnderReplicatedBlocks` 类小节。`UnderReplicatedBlocks.add()` 方法的实现也比较简单，根据数据块当前有的副本数量、撤销的副本数量以及期望的副本数量，确定这个副本复制的优先级，然后加入指定的优先级队列中。

```
synchronized boolean add(Block block,
                        int curReplicas,
                        int decommissionedReplicas,
                        int expectedReplicas) {
    assert curReplicas >= 0 : "Negative replicas!";
    int priLevel = getPriority(block, curReplicas, decommissionedReplicas,
                             expectedReplicas);
    if(priLevel != LEVEL && priorityQueues.get(priLevel).add(block)) {
        return true;
    }
    return false;
}
```

更新数据块复制操作

数据块复制操作的更新是通过 `BlockManager.updateNeededReplications()->UnderReplicatedBlocks.update()` 方法进行的，如图 3-44 所示，当 HDFS 中新添加了数据块副本、删除了数据块副本、数据块副本损坏以及重置了副本系数时，都会调用 `BlockManager.updateNeeded`

Replications()方法对 neededReplications 队列进行更新（例如添加和删除副本）。

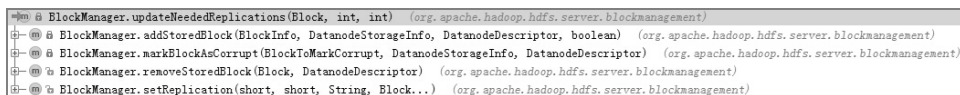


图 3-44 BlockManager.updateNeededReplications()调用关系

updateNeededReplications()方法首先会判断当前数据块的副本系数是否足够，如果不够则调用 neededReplications.update()方法更新 neededReplications 当前数据块的优先级。如果副本系数足够，也就是当前数据块不再需要执行复制操作了，updateNeededReplications()方法会调用 neededReplications.remove()从 neededReplications 队列中删除当前数据块。

```

private void updateNeededReplications(final Block block,
    final int curReplicasDelta, int expectedReplicasDelta) {
    namesystem.writeLock();
    try {
        if (!namesystem.isPopulatingReplQueues()) {
            return;
        }
        NumberReplicas repl = countNodes(block);
        int curExpectedReplicas = getReplication(block);
        // 判断当前数据块的有效副本系数是否足够，如果不够则更新 neededReplications
        if (isNeededReplication(block, curExpectedReplicas, repl.liveReplicas())) {
            neededReplications.update(block, repl.liveReplicas(), repl
                .decommissionedReplicas(), curExpectedReplicas, curReplicasDelta,
                expectedReplicasDelta);
        } else {
            // 当前数据块的副本系数已经满足了期望，那么从 neededReplications 中删除数据块
            int oldReplicas = repl.liveReplicas() - curReplicasDelta;
            int oldExpectedReplicas = curExpectedReplicas - expectedReplicasDelta;
            neededReplications.remove(block, oldReplicas, repl.decommissionedReplicas(),
                oldExpectedReplicas);
        }
    } finally {
        namesystem.writeUnlock();
    }
}
  
```

UnderReplicatedBlocks.update()方法的实现非常类似于 add()方法，首先根据参数计算出数据块原有的优先级，然后根据参数计算出数据块更新以后的优先级，最后更新当前副本的优先级队列。

```

synchronized void update(Block block, int curReplicas,
    int decommissionedReplicas,
    int curExpectedReplicas,
    int curReplicasDelta, int expectedReplicasDelta) {
    int oldReplicas = curReplicas - curReplicasDelta;
    int oldExpectedReplicas = curExpectedReplicas - expectedReplicasDelta;
  
```

```
int    curPri    =    getPriority(block,    curReplicas,    decommissionedReplicas,
curExpectedReplicas);
int    oldPri    =    getPriority(block,    oldReplicas,    decommissionedReplicas,
oldExpectedReplicas);

if(oldPri != LEVEL && oldPri != curPri) {
    remove(block, oldPri);
}
if(curPri != LEVEL && priorityQueues.get(curPri).add(block)) {
}
}
```

3. 块汇报

我们知道 Namenode 中数据块与数据节点的对应关系并不持久化在 fsimage 文件中, 而是由 Datanode 定期块汇报到 Namenode, 然后由 Namenode 重建内存中数据块与数据节点的对应关系。通过前面小节的学习我们知道, 数据块与数据节点存储的对应关系是由 BlockManager.blocksMap 字段维护的, 而数据节点存储与数据块的信息则是由 DatanodeStorageInfo.blockList 对象维护的。

Datanode 启动后, 会与 Namenode 握手、注册以及向 Namenode 发送第一次全量块汇报, 全量块汇报中包含了 Datanode 上存储的所有副本信息。之后, Datanode 的 BPSERVICEActor 对象会以 dfs.blockreport.intervalMsec (默认是 6 个小时) 间隔向 Namenode 发送全量块汇报, 同时会以 100*heartBeatInterval (心跳间隔的 100 倍, 默认为 300 秒) 间隔向 Namenode 发送增量块汇报, 增量块汇报中包含了 Datanode 最近新添加的以及删除的副本信息。

既然已经有了启动时的全量块汇报以定期发送的增量块汇报, 为什么还需要周期性地发送全量块汇报呢? 这是为了在 Datanode 的增量块汇报发生异常, 或者 Namenode 下发的复制或或删除副本指令丢失等情况发生时, Namenode 能够获取 Datanode 保存的所有副本的信息并执行相应的补救操作。

为了提高 HDFS 的启动速度, Namenode 会将 Datanode 的全量块汇报分为两种: 启动时发送的第一次全量块汇报和周期性的全量块汇报。对于启动时发送的第一次全量块汇报, 为了提高响应速度, Namenode 不会计算哪些元数据需要删除, 不会计算无效副本, 将这些处理都推迟到下一次块汇报时处理。

块汇报操作到达 Namenode 之后会由 BlockManager.processReport() 方法响应。processReport() 方法会判断当前块汇报是否是该数据节点的第一次块汇报, 如果是则调用 processFirstBlockReport() 方法处理, 这个方法的效率会很高。如果不是第一次块汇报, 则调用私有的 proceeReport() 方法处理。之后, processReport() 方法还会对 postponedMisreplicatedBlock 集合重新扫描, 删除那些已经不是 stale 状态的副本。

```
public boolean processReport(final DatanodeID nodeID,
    final DatanodeStorage storage,
    final BlockListAsLongs newReport) throws IOException {
    // ...
```



```

if (node.numBlocks() == 0) {
    // 对于第一次块汇报，调用 processFirstBlockReport()
    processFirstBlockReport(node, newReport);
} else {
    // 不是第一次块汇报，则调用私有的 processReport() 方法
    processReport(node, newReport);
}

// ...

boolean staleBefore = node.areBlockContentsStale();
node.receivedBlockReport();
// 如果节点之前是 stale 状态，由于发起了新的 BlockReport，则该节点脱离 stale 状态
if (staleBefore && !node.areBlockContentsStale()) {
    rescanPostponedMisreplicatedBlocks();
}
// ...
}

```

下面我们学习 processFirstBlockReport()方法以及私有的 processReport()方法的实现。

（1）第一次块汇报——processFirstBlockReport()

Datanode 会通过调用 NamenodeProtocol.blockReport()方法向 Namenode 发送全量块汇报，请求到达 Namenode 后的代码调用流程为 NameNodeRpcServer.blockReport()->BlockManager.processReport()。processReport()方法会调用 processFirstBlockReport()来处理 Datanode 启动后的第一次全量块汇报，processFirstBlockReport()方法会调用 addStoredBlockImmediate()方法将块汇报中所有有效的副本加入 Namenode 内存中，之后 processFirstBlockReport()方法会调用 markBlockAsCorrupt()方法处理无效副本（Datanode 上存在、Namenode 的 blocksMap 中不存在），注意 processFirstBlockReport()方法并不处理需要删除的副本（Datanode 上不存在、Namenode 内存中存在）。要特别注意的是，在 HDFS HA 架构中，Datanode 的心跳信息、全量块汇报以及增量块汇报会同时发送到 Standby Namenode 以及 Active Namenode。StandBy Namenode 处理全量块汇报时，可能出现命名空间还未与 Active Namenode 同步的情况，这时就需要将待处理副本暂时缓存起来，等待 StandBy Namenode 完全加载 editlog 并更新命名空间后再处理。processFirstBlockReport()方法的代码如下：

```

BlockReportIterator itBR = report.getBlockReportIterator();
while(itBR.hasNext()) {
    // 首先获取块汇报中副本的状态 (ReplicaState)
    Block iblk = itBR.next();
    ReplicaState reportedState = itBR.getCurrentReplicaState();

    // 如果当前 Namenode 为 Standby Namenode，那么可能出现块汇报时 Standby Namenode 没有完全
    更新 editlog 的情况，也就是 Standby Namenode 的命名空间不是最新的，这时就需要等待 Standby Namenode
    更新最新的 editlog
    if (shouldPostponeBlocksFromFuture &&

```

```
        namesystem.isGenStampInFuture(iblk)) {
            queueReportedBlock(storageInfo, iblk, reportedState,
                QUEUE_REASON_FUTURE_GENSTAMP);
            continue;
        }
        BlockInfo storedBlock = blocksMap.getStoredBlock(iblk);
        // 如果汇报的数据块在Namenode的blocksMap中并不存在,则放弃处理
        if (storedBlock == null) continue;
        // 判断当前数据块是否是损坏的数据块 (corrupt)
        BlockUCState ucState = storedBlock.getBlockUCState();
        BlockToMarkCorrupt c = checkReplicaCorrupt(
            iblk, reportedState, storedBlock, ucState,
            storageInfo.getDatanodeDescriptor());
        if (c != null) {
            if (shouldPostponeBlocksFromFuture) {
                // 同样需要考虑 Standby 节点的情况
                queueReportedBlock(storageInfo, iblk, reportedState,
                    QUEUE_REASON_CORRUPT_STATE);
            } else {
                // 将损坏的节点放入 invalidateBlocks 队列中
                markBlockAsCorrupt(c, storageInfo, storageInfo.getDatanodeDescriptor());
            }
            continue;
        }

        // 如果数据块处于构建状态,那么将这个副本加入对应的构建中副本队列中
        if (isBlockUnderConstruction(storedBlock, ucState, reportedState)) {
            ((BlockInfoUnderConstruction)storedBlock).addReplicaIfNotPresent(
                storageInfo, iblk, reportedState);
            BlockInfoUnderConstruction blockUC = (BlockInfoUnderConstruction) storedBlock;
            if (namesystem.isInSnapshot(blockUC)) {
                int numOfReplicas = blockUC.getNumExpectedLocations();
                namesystem.incrementSafeBlockCount(numOfReplicas);
            }
        }
        // 对于正常状态的副本,调用 addStoredBlockImmediate() 加入 Namenode 内存中
        if (reportedState == ReplicaState.FINALIZED) {
            addStoredBlockImmediate(storedBlock, storageInfo);
        }
    }
}
```

`addStoredBlockImmediate()`是 `addStoredBlock()`的快速版本,用于在 Namenode 内存中添加一个新的数据块副本。如果 Namenode 当前处于安全模式下,则执行快速添加操作,否则还是调用普通的 `addStoredBlock()`方法, `addStoredBlock()`方法的实现请参考添加副本小节。快速版本的 `addStoredBlockImmediate()`方法并不考虑 `underReplication`、`overReplication`、`pendingReplications`、`corruptReplicas` 等队列的更新操作,也不用记录日志,而是直接在内存中添加这个副本的信息。因为刚启动的 Namenode 会有数以万计的数据块副本需要处理,所

以需要快速地将它们加入 BlockManager 的内存中。

```
private void addStoredBlockImmediate(BlockInfo storedBlock,
    DatanodeStorageInfo storageInfo)
throws IOException {
    assert (storedBlock != null && namesystem.hasWriteLock());
    if (!namesystem.isInStartupSafeMode()
        || namesystem.isPopulatingReplQueues()) {
        addStoredBlock(storedBlock, storageInfo, null, false);
        return;
    }

    // 在 Namenode 内存中添加这个数据块副本的信息
    storageInfo.addBlock(storedBlock);

    // 如果当前数据块处于 COMMITTED 状态，并且数据块满足最小的副本要求
    // 则调用 completeBlock() 方法将数据块的状态更新为 COMPLETE
    int numCurrentReplica = countLiveNodes(storedBlock);
    if (storedBlock.getBlockUCState() == BlockUCState.COMMITTED
        && numCurrentReplica >= minReplication) {
        completeBlock(storedBlock.getBlockCollection(), storedBlock, false);
    } else if (storedBlock.isComplete()) {
        namesystem.incrementSafeBlockCount(numCurrentReplica);
    }
}
```

addStoredBlockImmediate()方法会调用 DatanodeStorageInfo.addBlock()将数据块副本的信息加入 Namenode 内存中。addBlock()方法首先会更新副本对应的 BlockInfo 对象的 triplets[] 数组，将当前数据块存储对应的 DatanodeStorageInfo 对象加入 triplets[] 数组中，然后将副本对应的 BlockInfo 添加到 DatanodeStorageInfo 的 blockList 队列中。addBlock()方法更新了数据块存储与副本的对应关系，同时还更新了数据块存储与存储上保存的副本的对应关系。

（2）普通块汇报——processReport()

对于 Datanode 周期性的块汇报，processReport()方法会调用私有的 processReport()方法处理。这个方法会调用 reportDiff()方法，将块汇报中的副本与当前 Namenode 内存中记录的副本状态做比对，然后产生 5 个操作队列。

- toAdd——上报副本与 Namenode 内存中记录的数据块有相同的时间戳以及长度，那么将上报副本添加到 toAdd 队列中。对于 toAdd 队列中的元素，调用 addStoredBlock()方法将副本添加到 Namenode 内存中。对 addStoredBlock()方法的分析请参考添加副本小节。
- toRemove——副本在 Namenode 内存中的 DatanodeStorageInfo 对象上存在，但是块汇报时并没有上报该副本，那么将副本添加到 toRemove 队列中。对于 toRemove 队列中的元素，调用 removeStoredBlock()方法将数据块从 Namenode 内存中删除。对 removeStoredBlock()方法的分析请参考删除副本小节。

- **toInvalidate**——BlockManager 的 blocksMap 字段中没有保存上报副本的信息，那么将上报副本添加到 toInvalidate 队列中。对于 toInvalidate 队列中的元素，调用 addToInvalidates() 方法将该副本加入 BlockManager.invalidateBlocks 队列中，然后触发 Datanode 节点删除该副本。
- **toCorrupt**——上报副本的时间戳或者文件长度不正常，那么将上报副本添加到 corruptReplicas 队列中。对于 corruptReplicas 队列中的元素，调用 markBlockAsCorrupt() 方法处理。对 markBlockAsCorrupt() 方法的分析请参考损坏副本的删除小节。
- **toUC**——如果上报副本对应的数据块处于构建状态，则调用 addStoredBlockUnderConstruction() 方法构造一个 ReplicateUnderConstruction 对象，然后将该对象添加到数据块对应的 BlockInfoUnderConstruction 对象的 replicas 队列中。

processReport() 方法的代码如下所示：

```
private void processReport(final DatanodeDescriptor node,
    final BlockListAsLongs report) throws IOException {

    // 调用 reportDiff() 方法获取需要更新的不同队列
    Collection<BlockInfo> toAdd = new LinkedList<BlockInfo>();
    Collection<Block> toRemove = new LinkedList<Block>();
    Collection<Block> toInvalidate = new LinkedList<Block>();
    Collection<BlockToMarkCorrupt> toCorrupt = new LinkedList<BlockToMarkCorrupt>();
    Collection<StatefulBlockInfo> toUC = new LinkedList<StatefulBlockInfo>();
    reportDiff(node, report, toAdd, toRemove, toInvalidate, toCorrupt, toUC);

    // 调用对应方法处理不同的队列
    for (StatefulBlockInfo b : toUC) {
        addStoredBlockUnderConstruction(b.storedBlock, node, b.reportedState);
    }
    for (Block b : toRemove) {
        removeStoredBlock(b, node);
    }
    for (BlockInfo b : toAdd) {
        addStoredBlock(b, node, null, true);
    }
    for (Block b : toInvalidate) {
        addToInvalidates(b, node);
    }
    for (BlockToMarkCorrupt b : toCorrupt) {
        markBlockAsCorrupt(b, node);
    }
}
```

下面我们看一下 reportDiff() 方法的实现。reportDiff() 首先添加一个分隔节点 delimiter 到 DatanodeStorageInfo 的 blockList 的头部，然后遍历块汇报中的所有副本并调用 processReportedBlock() 方法更新 toAdd、toInvalidate、toCorrupt、toUC 队列。接下来 reportDiff() 将处理完的副本移动到 blockList 的头部，这样 delimiter 就分隔了 blockList 中块汇报包含的副

本和没有包含的副本。最后 `reportDiff()` 会将没有汇报的副本加入 `toRemove` 队列中。`reportDiff()` 方法的代码如下：

```
private void reportDiff(DatanodeStorageInfo storageInfo,
    BlockListAsLongs newReport,
    Collection<BlockInfo> toAdd,
    Collection<Block> toRemove,
    Collection<Block> toInvalidate,
    Collection<BlockToMarkCorrupt> toCorrupt,
    Collection<StatefulBlockInfo> toUC) {

    // 这里添加一个分隔 Block 到 blockList 的头部，将 blockList 中块汇报包含的副本和没有包含的副本分
    隔开来
    BlockInfo delimiter = new BlockInfo(new Block(), 1);
    boolean added = storageInfo.addBlock(delimiter);
    int headIndex = 0; // 当前分隔符在 blockList 的头部
    int curIndex;

    if (newReport == null) {
        newReport = new BlockListAsLongs();
    }

    BlockReportIterator itBR = newReport.getBlockReportIterator();
    while(itBR.hasNext()) {
        Block iblk = itBR.next();
        ReplicaState iState = itBR.getCurrentReplicaState();
        // 调用 processReportedBlock() 更新 toAdd、toInvalidate、toCorrupt、toUC 操作队列
        BlockInfo storedBlock = processReportedBlock(storageInfo,
            iblk, iState, toAdd, toInvalidate, toCorrupt, toUC);

        // 将处理过的副本移动到 blockList 的头部
        if (storedBlock != null &&
            (curIndex = storedBlock.findStorageInfo(storageInfo)) >= 0) {
            headIndex = storageInfo.moveBlockToHead(storedBlock, curIndex, headIndex);
        }
    }

    // 所有分隔符后的副本为没有上报的副本，但是这些副本存在于 Namenode 中，所以直接加入 toRemove 队列
    中，之后从 Namenode 内存中删除
    Iterator<BlockInfo> it = storageInfo.new BlockIterator(delimiter.getNext(0));
    while(it.hasNext())
        toRemove.add(it.next());
    storageInfo.removeBlock(delimiter);
}
```

`reportDiff()` 方法调用了 `processReportedBlock()` 方法处理 `toAdd`、`toInvalidate`、`toCorrupt`、`toUC` 这几个队列。我们分段看一下 `processReportedBlock()` 的实现。

- 如果上报副本在 `BlockManager.blocksMap` 中不存在，则将这个副本加入 `toInvalidate` 队列中。

```
BlockInfo storedBlock = blocksMap.getStoredBlock(block);
if(storedBlock == null) {
    toInvalidate.add(new Block(block));
    return null;
}
```

- 调用 `checkReplicaCorrupt()` 根据上报副本的状态以及该副本对应的数据块状态确定该副本是否为损坏的副本，如果该副本已经损坏，则将该副本添加到 `toCorrupt` 队列中。

```
BlockToMarkCorrupt c = checkReplicaCorrupt(
    block, reportedState, storedBlock, ucState, dn);
if (c != null) {
    if (shouldPostponeBlocksFromFuture) {
        // 当前 Namenode 处于 Standby 情况时，推迟处理
        queueReportedBlock(dn, storedBlock, reportedState,
            QUEUE_REASON_CORRUPT_STATE);
    } else {
        toCorrupt.add(c);
    }
    return storedBlock;
}
```

- 调用 `isBlockUnderConstruction()` 方法判断当前上报副本对应的数据块是否处于构建状态，如果是则加入 `toUC` 队列中。

```
if (isBlockUnderConstruction(storedBlock, ucState, reportedState)) {
    toUC.add(new StatefulBlockInfo(
        (BlockInfoUnderConstruction)storedBlock, reportedState));
    return storedBlock;
}
```

- 如果上报副本有效，并且在 `BlockManager.blocksMap` 中没有保存这个副本的信息，则加入 `toAdd` 队列中。

```
if (reportedState == ReplicaState.FINALIZED
    && (storedBlock.findStorageInfo(storageInfo) == -1 ||
        corruptReplicas.isReplicaCorrupt(storedBlock, dn))) {
    toAdd.add(storedBlock);
}
```

这里我们需要讲解 `checkReplicaCorrupt()` 和 `isBlockUnderConstruction()` 方法，这两个方法会根据当前汇报副本的状态以及该副本对应的数据块状态确定该副本是否损坏，或者处于构建中。

`checkReplicaCorrupt()` 方法会对块汇报副本与该副本对应数据块的时间戳、长度进行比较，如果有某项不一致，就认为该副本已经损坏。表 3-1 给出了 `checkReplicaCorrupt()` 方法的判断逻辑。

表 3-1 checkReplicaCorrupt()方法的判断逻辑

Datanode 报告的副本状态	FINALIZED	FINALIZED	RBW / RWR	RUR/TEMPORARY
Namenode 记录数据块状态	COMMITTED COMPLETE	UNDER_CONSTRUCTION	COMPLETE	任意状态
判断条件	时间戳与副本长度不一致,则为损坏的数据块	Namenode 上记录的构建中数据块的时间戳大于 Datanode 上持久化的副本的时间戳,这种情况不可能出现,标识为损坏的数据块	时间戳不一致,则标识为损坏的数据块	这两个状态的副本不应该出现在块汇报中,标识为损坏节点

- 对于汇报副本是 FINALIZED 状态, Namenode 记录数据块是 COMMITTED 和 COMPLETE 状态, 如果时间戳以及副本长度不一致, 则表明数据块损坏了。
- 对于汇报副本是 FINALIZED 状态, Namenode 记录数据块是 UNDER_CONSTRUCTION 状态, 如果 Namenode 记录的时间戳大于 Datanode 汇报的, 则表明数据块已经损坏。这种情况很有可能是客户端对当前数据块发起了追加写操作, Namenode 上数据块的时间戳递增, 但是 Datanode 由于故障并没有接收到追加写请求, 也没有更新数据块的状态与时间戳。
- 如果汇报副本的状态是 RBW / RWR, Namenode 记录数据块为 COMPLETE 状态, 且时间戳不一致, 则表明数据块已经损坏。这种情况有可能是数据块的数据流管道在传输数据流时出现异常, 某个 Datanode 仅接收了部分数据块信息, 所以与 Namenode 的时间戳不一致。
- 如果汇报副本的状态为 RUR/TEMPORARY, 则认为该副本已经损坏, 因为在正常情况下, 这两个状态的副本不应该出现在块汇报中。

isBlockUnderConstruction()也会根据汇报副本的状态以及 Namenode 中数据块的状态判断当前副本是否作为构建中副本处理。isBlockUnderConstruction()方法的处理逻辑请参考表 3-2。

表 3-2 isBlockUnderConstruction()方法的处理逻辑

Datanode 报告的副本状态	FINALIZED	RBW/RWR	RUR/TEMPORARY
Namenode 记录数据块状态	UNDER_CONSTRUCTION UNDER_RECOVERY	不是 COMPLETE	任意状态
判断结果	true	true	false

(3) 增量块汇报——processIncrementalBlockReport()

Datanode 会调用 NamenodeProtocol.blockReceivedAndDeleted()方法将短时间内接收的副本或者删除的副本增量汇报给 Namenode, Namenode 收到了增量汇报后, 会调用 processIncrementalBlockReport()方法处理。processIncrementalBlockReport()方法会遍历增量汇报中的所有数据块, 如果是新添加的数据块 (RECEIVED_BLOCK), 则调用 addBlock()方法处理添加请求; 如果是删除的数据块 (DELETED_BLOCK), 则调用 removeStorageBlock()修改数据块与存储这个数据块的数据节点存储的对应关系。对于接收中的副本 (RECEIVING_

BLOCK), 则调用 `processAndHandleReportedBlock()` 方法处理。 `processIncrementalBlockReport()` 方法的代码如下:

```
public void processIncrementalBlockReport(final DatanodeID nodeID,
    final StorageReceivedDeletedBlocks srdb) throws IOException {
    // ...

    for (ReceivedDeletedBlockInfo rdbi : srdb.getBlocks()) {
        switch (rdbi.getStatus()) {
            case DELETED_BLOCK:
                removeStoredBlock(rdbi.getBlock(), node);
                deleted++;
                break;
            case RECEIVED_BLOCK:
                addBlock(storageInfo, rdbi.getBlock(), rdbi.getDelHints());
                received++;
                break;
            case RECEIVING_BLOCK:
                receiving++;
                processAndHandleReportedBlock(storageInfo, rdbi.getBlock(),
                    ReplicaState.RBW, null);

                break;
            default:
                break;
        }
    }
    // ...
}
```

对于增量汇报中新添加的副本, 可能是客户端通过输入流管道写入了一个副本, 也有可能是 **Namenode** 发起的复制操作。这里由 `addBlock()` 方法处理增量块汇报中新添加的数据块, `addBlock()` 方法会更改 `DatanodeDescriptor` 上的 `blockScheduled` 计数, 然后从 `pendingReplications` 中移除这个数据节点上该数据块的复制请求, 最后调用 `processAndHandleReportedBlock()` 处理副本为提交状态 (FINALIZED) 的数据块副本。

```
void addBlock(DatanodeStorageInfo storageInfo, Block block, String delHint)
    throws IOException {
    DatanodeDescriptor node = storageInfo.getDatanodeDescriptor();
    // 减少当前节点上的 blockScheduled 计数
    node.decrementBlocksScheduled(storageInfo.getStorageType());

    // 获取 delHintNode, 也就是希望从 delHintNode 中删除该数据块
    DatanodeDescriptor delHintNode = null;
    if (delHint != null && delHint.length() != 0) {
        delHintNode = datanodeManager.getDatanode(delHint);
    }

    // 如果是当前节点上安排的复制操作, 则从 pendingReplications 中移出该请求
    pendingReplications.decrement(block, node);
}
```



```
// 调用 processAndHandleReportedBlock() 方法处理这次添加请求
// 注意副本状态是 FINALIZED
processAndHandleReportedBlock(storageInfo, block, ReplicaState.FINALIZED,
    delHintNode);
}
```

下面看一下 `processAndHandleReportedBlock()` 的实现。对于新添加数据块以及接收中数据块这两种情况都调用了 `processAndHandleReportedBlock()` 方法，它调用了 `processReportedBlock()` 方法将这个单独的数据块添加到对应的处理队列中，然后调用相应的方法处理各个队列中的数据块，类似于上一节介绍的 `processReport()` 方法。

```
private void processAndHandleReportedBlock(
    DatanodeStorageInfo storageInfo, Block block,
    ReplicaState reportedState, DatanodeDescriptor delHintNode)
    throws IOException {
    Collection<BlockInfo> toAdd = new LinkedList<BlockInfo>();
    Collection<Block> toInvalidate = new LinkedList<Block>();
    Collection<BlockToMarkCorrupt> toCorrupt = new LinkedList<BlockToMarkCorrupt>();
    Collection<StatefulBlockInfo> toUC = new LinkedList<StatefulBlockInfo>();
    final DatanodeDescriptor node = storageInfo.getDatanodeDescriptor();

    processReportedBlock(storageInfo, block, reportedState,
        toAdd, toInvalidate, toCorrupt, toUC);

    for (StatefulBlockInfo b : toUC) {
        addStoredBlockUnderConstruction(b, storageInfo);
    }
    long numBlocksLogged = 0;
    for (BlockInfo b : toAdd) {
        addStoredBlock(b, storageInfo, delHintNode, numBlocksLogged < maxNumBlocksToLog);
        numBlocksLogged++;
    }
    if (numBlocksLogged > maxNumBlocksToLog) {
    }
    for (Block b : toInvalidate) {
        addToInvalidates(b, node);
    }
    for (BlockToMarkCorrupt b : toCorrupt) {
        markBlockAsCorrupt(b, storageInfo, node);
    }
}
```

对于增量汇报中的删除副本，`processIncrementalBlockReport()` 方法会调用 `removeStoredBlock()` 方法执行删除操作。`removeStoredBlock()` 方法的实现请参考删除数据块小节。

3.3 数据节点管理

我们知道名字节点启动之后，会加载 `fsimage` 和 `editlog` 文件重建文件系统目录树，但是

对于数据块与 Datanode 的映射关系却需要在 Datanode 上报后动态构建。Datanode 在启动时除了会与名字节点握手、注册以及上报数据块信息外，还会定时向 Namenode 发送心跳以及块汇报，并执行 Namenode 传回的指令。所以 Namenode 中会有很大一部分逻辑是与 Datanode 相关的，包括添加和删除 Datanode、与 Datanode 启动过程的交互、处理 Datanode 发送的心跳。本节就重点介绍 Namenode 是如何管理 Datanode 的。

3.3.1 DatanodeDescriptor

数据节点描述符（DatanodeDescriptor）是 Namenode 中对 Datanode 的抽象，继承自 DatanodeInfo 类。DatanodeDescriptor 的类继承结构如图 3-45 所示。

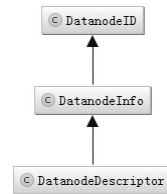


图 3-45 DatanodeDescriptor 类继承关系图

下面我们依次看一下图 3-45 中各个类的实现。

1. DatanodeId

DatanodeID 用于唯一标识一个 Datanode，Datanode 是通过<ip, port>以及 storageID 进行标识的，如下代码所示。

```
private String ipAddr;      // IP 地址
private String hostName;    // Datanode 声明的 hostname
private String peerHostName; // 真实连接的 hostname
private String storageID;   // storageID 用于唯一标识 Datanode，是集群唯一的
private int xferPort;       // 数据传输端口
private int infoPort;       // info 服务端口
private int infoSecurePort; // info 服务端口
private int ipcPort;        // IPC 服务端口
```

2. DatanodeInfo

DatanodeInfo 类扩展自 DatanodeId，它携带了一些比较简单的 Datanode 信息，如下代码所示。

```
private long capacity; // 容量
private long dfsUsed;   // 使用的空间
private long remaining; // 剩余空间
private long blockPoolUsed; // 数据块池使用量
private long cacheCapacity; // 缓存容量
private long cacheUsed; // 缓存使用量
private long lastUpdate; // 上次更新时间
private int xceiverCount; // xceiver 数量
private String location = NetworkTopology.DEFAULT_RACK; // 地址
private String softwareVersion; // 软件版本
```

还有一个比较重要的字段——adminState，用标识当前 Datanode 可能处于的状态。HDFS 使用 AdminStates 定义了 Datanode 可能处于的状态，如下代码所示。

```
protected AdminStates adminState;

public enum AdminStates {
    NORMAL("In Service"),
    DECOMMISSION_INPROGRESS("Decommission In Progress"),
    DECOMMISSIONED("Decommissioned");

    final String value;
    // ...
}
```

- **NORMAL**: Datanode 处于正常服务状态。
- **DECOMMISSION_INPROGRESS**: Datanode 处于撤销状态中（通过 admin 命令将指定 Datanode 撤销）。
- **DECOMMISSIONED**: 已经撤销。

3. DatanodeDescriptor

DatanodeDescriptor 是 Namenode 中用于描述一个 Datanode 信息的类，这个类非常重要。需要特别注意，这个类只用在 Namenode 侧，对于 Client 是不可见的。DatanodeDescriptor 继承自 DatanodeInfo 类。

DatanodeDescriptor 类定义了很多字段，如图 3-46 所示，我们只讲解当中比较重要的几个字段。

```

- ① decommissioningStatus: DecommissioningStatus = new DecommissioningStatus()
- ② isAlive: boolean = false
- ③ needKeyUpdate: boolean = false
- ④ storageMap: Map<String, DatanodeStorageInfo> = new HashMap<String, DatanodeStorageInfo>()
- ⑤ pendingCached: CachedBlocksList = new CachedBlocksList(...)
- ⑥ cached: CachedBlocksList = new CachedBlocksList(...)
- ⑦ pendingUncached: CachedBlocksList = new CachedBlocksList(...)
- ⑧ lastCachingDirectiveSentTimeMs: long
- ⑨ bandwidth: long
- ⑩ replicateBlocks: BlockQueue<BlockTargetPair> = new BlockQueue<BlockTargetPair>()
- ⑪ recoverBlocks: BlockQueue<BlockInfoUnderConstruction> = new BlockQueue<BlockInfoUnderConstruction>()
- ⑫ invalidateBlocks: LightweightHashSet<Block> = new LightweightHashSet<Block>()
- ⑬ currApproxBlocksScheduled: EnumCounters<StorageType> = new EnumCounters<StorageType>(...)
- ⑭ prevApproxBlocksScheduled: EnumCounters<StorageType> = new EnumCounters<StorageType>(...)
- ⑮ lastBlocksScheduledRollTime: long = 0
- ⑯ BLOCKS_SCHEDULED_ROLL_INTERVAL: int = 600 * 1000
- ⑰ volumeFailures: int = 0
- ⑱ disallowed: boolean = false
- ⑲ heartbeatedSinceRegistration: boolean = false
- ⑳ PendingReplicationWithoutTargets: int = 0

```

图 3-46 DatanodeDescriptor 字段

- **Datanode 状态相关**: 包括 isAlive、decommissioningStatus、currApproxBlocksScheduled 等字段。isAlive 记录当前 Datanode 是否有效；decommissioningStatus 记录撤销操作时节点的状态（只在 Datanode 处于撤销状态时使用）；currApproxBlocksScheduled 等几个成员变量用于估计 Datanode 的负载，在写文件操作分配数据块或进行数据块复制时，根据节点负载信息，选择比较空闲的节点作为目标节点。
- **指令相关**: DatanodeDescriptor 中的 bandwidth、replicateBlocks、recoverBlocks、invalidateBlocks 字段都是与 Namenode 向 Datanode 下达指令相关的字段。

- **bandwidth**: HDFS 系统管理员可以通过 “`dfsadmin -setBalancerBandwidth <newbandwidth>`” 命令更改均衡器 (balancer) 带宽。新设置的带宽值将会赋值给 **bandwidth** 字段, 并通过心跳下发到指定 **Datanode**。当下发指令完成后, **bandwidth** 将会被重置为 0。
- **replicateBlocks**: 保存要在当前 **Datanode** 上复制的副本队列, **Namenode** 会在下次心跳回复中将这个队列中的一部分副本带到 **Datanode**, 并下发复制指令, **Datanode** 接受指令后会将该数据块副本复制到目标数据节点。**replicateBlocks** 的类型是 **BlockQueue**, 它保存的元素类型是 **BlockTargetPair**。**BlockQueue** 是 **DatanodeDescriptor** 的内部类, 是一个简单的队列类。**BlockTargetPair** 类则包含了要被复制的数据块以及复制操作的目标数据节点 (列表) 两项信息。
- **recoverBlocks**: 保存要在当前 **Datanode** 上进行数据块恢复操作的副本队列, **Namenode** 会在下次心跳回复中将这个队列中的一部分副本带到 **Datanode**, 并下发数据块恢复指令, **Datanode** 接受指令后会作为主恢复数据节点指导整个数据块的恢复操作 (请参考租约管理的租约恢复小节)。与 **replicateBlocks** 类似, **recoverBlocks** 的类型也是保存 **BlockTargetPair** 的 **BlockQueue**, 这里 **BlockTargetPair** 的目标数据节点是所有参与恢复过程的 **Datanode**。
- **invalidateBlocks**: 要在 **Datanode** 上进行删除操作的副本队列。
- **缓存相关**: 包括 **pendingCached**、**cached**、**pendingUncached** 三个字段, 它们都是 **CachedBlocksList** 类型的, **CachedBlocksList** 是 **CachedBlock** 的集合类, **CachedBlock** 类则用于描述一个被缓存的数据块。**pendingCached** 字段中保存的是所有在当前 **Datanode** 上等待缓存的数据块; **cached** 中保存的是当前 **Datanode** 上已经缓存的数据块, **pendingUncached** 中保存的则是等待取消缓存的数据块。

DatanodeDescriptor 定义的方法也都比较简单, 大多是对上面描述的几个比较重要字段的增、删、改、查操作, 这里就不再赘述了, 感兴趣的读者可以自己阅读。

3.3.2 DatanodeStorageInfo

DatanodeStorageInfo 类描述了 **Datanode** 上的一个存储 (storage), 一个 **Datanode** 可以定义多个存储 (在 `dfs.datanode.data.dir` 中配置多个 **Datanode** 的存储目录) 来保存数据块, 这些存储还可以是异构的, 例如可以是磁盘、内存、SSD 等。需要特别注意的是, 在 HDFS 2.6 版本前, **Namenode** 内存中维护的第二关系是数据块与保存数据块副本的数据节点的对应关系, 也就是 **Block** 与 **DatanodeDescriptor** 的对应关系。在 HDFS 2.6 版本中, 为了支持 **Datanode** 的异构存储特性, **Namenode** 内存中维护的第二关系变成了数据块与保存数据块副本的数据节点存储的对应关系, 也就是 **Block** 与 **DatanodeStorageInfo** 的对应关系, 同时块汇报的单位也由 **Datanode** 变为了 **DatanodeStorageInfo**。

DatanodeStorageInfo 类定义的字段如图 3-47 所示。

```

① @ dn: DatanodeDescriptor
② @ storageID: String
③ @ storageType: StorageType
④ @ state: State
⑤ @ capacity: long
⑥ @ dfsUsed: long
⑦ @ remaining: long
⑧ @ blockPoolUsed: long
⑨ @ blockList: BlockInfo = null
⑩ @ numBlocks: int = 0
⑪ @ blockReportCount: int = 0
⑫ @ heartbeatedSinceFailover: boolean = false
⑬ @ blockContentsStale: boolean = true

```

图 3-47 DatanodeStorageInfo 字段

- **BlockInfo blockList:** DatanodeStorageInfo 中最重要的数据结构，用来记录当前存储上保存的数据块副本链表的头节点。回忆 BlockInfo 类的 triplets[] 数组记录了当前存储保存的前一个数据块以及后一个数据块，是一个轻量级的双向链表。当 Datanode 报告 Namenode 成功地接收了一个数据块副本后，Namenode 会调用 DatanodeStorageInfo.addBlock() 方法在该 DatanodeStorageInfo 的 blockList 中添加这个副本对应的 BlockInfo 对象。
- **Namenode 状态相关:** heartbeatedSinceFailover 和 blockContentsStale 两个字段用于记录 Namenode 发生错误切换时的信息。
 - **heartbeatedSinceFailover:** 当 Namenode 出现失败时，会将这个字段设置为 false；当 Namenode 正常接收到这个存储的心跳后，会将这个字段设置为 true。
 - **blockContentsStale:** 当 Namenode 出现失败或者正在启动时，Datanode 会挂起上一次 Namenode 发起的删除操作。这个时候我们就认为当前存储为 stale 状态，直到 Namenode 收到了这个存储的块汇报。当一个存储处于 stale 状态时，这个存储上的所有副本都是 stale 状态的，如果一个数据块至少有一个 stale 副本，那么这个数据块也为 stale 状态，stale 状态数据块的所有副本是不可以执行删除操作的（HDFS-1972）。

考虑下面的情况，当 Namenode 发生失败，Active Namenode 和 Standby Namenode 切换时，上一次进行副本删除操作的 Datanode 存储还没有进行块汇报（Datanode 存储可能完成了删除操作，也有可能没有完成）。在这种情况下，原来的 Standby Namenode 中的数据块信息就很有可能不完整，如果这时再次对这个数据块的副本发起删除操作，就很有可能丢失这个数据块（例如数据块的副本数为 2 时，Datanode1 已经删除了一个副本，但是没有汇报。这时如果 Namenode 再次发起删除操作，触发了副本 2 的删除操作，数据块的两个副本全被删除，这个数据块也就丢失了）。

所以现在 HDFS 的逻辑是当出现 Active 与 Standby 切换时，Namenode 会将所有 Datanode 存储的 heartbeatedSinceFailover 以及 blockContentsStale 字段设置为 false 和 true。然后扫描内存中的所有数据块，如果当前数据块有副本为 stale 状态，那么就将这个数据块放入 postponedMisreplicatedBlocks 队列中，直到所有 stale 状态的 Datanode 存储进行了块汇报。

- **storage 元信息:** dn、storageId、storageType、state 等字段用于描述当前存储的元信息。其中 dn 字段是当前存储所在 Datanode 对应的 DatanodeDescriptor 对象；storageId

是该存储在集群内唯一的标识符；storageType 用于描述当前存储是什么类型，例如是磁盘还是闪存等。

- storage 状态信息：包括 capacity、dfsUsed、remaining、blockPoolUsed、numBlocks 等信息，分别用于描述当前存储的容量、使用量、剩余容量、块池的使用量以及当前存储中保存的副本数量。

下面我们学习 DatanodeStorageInfo 的方法。当 Datanode 向 Namenode 汇报该 Datanode 存储上接收了一个新的数据块副本时，BlockManager 会调用 addBlock()方法在 Namenode 的第二关系中添加这个副本与保存副本的 Datanode 存储的对应关系。addBlock()方法首先调用 BlockInfo.addStorage()将自己也就是 DatanodeStorageInfo 对象添加到数据块所属的 Datanodes 存储列表中，也就是数据块对应的 BlockInfo 对象的 triplets[] 数组中。然后调用 BlockInfo.listInsert()将数据块插入到 Datanode 存储管理的数据块链表中，也就是将 BlockInfo 对象加入 Datanode 存储对应的 DatanodeStorageInfo 对象的 blockList 链表中。

```
public boolean addBlock(BlockInfo b) {
    // 首先检查这个数据块是否属于同一个 Datanode 上的另一个存储
    boolean replaced = false;
    DatanodeStorageInfo otherStorage =
        b.findStorageInfo(getDatanodeDescriptor());

    if (otherStorage != null) {
        if (otherStorage != this) {
            // 如果当前数据块属于另一个存储，则先从该存储上删除这个数据块
            otherStorage.removeBlock(b);
            replaced = true;
        } else {
            // 数据块已经添加到了当前存储上，不需要再次添加
            return false;
        }
    }

    // 首先将当前存储添加到数据块所属的存储列表中
    b.addStorage(this);
    // 之后将当前数据块添加到存储管理的数据块链表中
    blockList = b.listInsert(blockList, this);
    numBlocks++;
    return !replaced;
}
```

BlockInfo 的 addStorage()方法就是在 triplets[]数组中找到插入当前 DatanodeStorageInfo 的位置，并插入。listInsert()方法则将当前数据块对应的 BlockInfo 对象添加到 Datanode 存储管理的数据块链表中，插入的方法是直接插入在链表的头节点 blockList 之前，并用新添加数据块的 BlockInfo 对象替代原有的 blockList 作为数据块链表的头节点。

```
public class BlockInfo extends Block implements LightweightGSet.LinkedElement {
    BlockInfo listInsert(BlockInfo head, DatanodeStorageInfo storage) {
        int dnIndex = this.findStorageInfo(storage);
```

```

    this.setPrevious(dnIndex, null);    // 设置当前 BlockInfo 对象为链表的头节点
    this.setNext(dnIndex, head);
    if(head != null)
        head.setPrevious(head.findStorageInfo(storage), this);
    return this;                        // 返回当前节点作为链表的头节点
}

boolean addStorage(DatanodeStorageInfo storage) {
    int lastNode = ensureCapacity(1); // 在 triplets[] 数组中找到当前存储的插入位置
    setStorageInfo(lastNode, storage); // 插入 DatanodeStorageInfo 对象
    setNext(lastNode, null);
    setPrevious(lastNode, null);
    return true;
}
}

```

我们再来看一下 `markStaleAfterFailover()` 方法，这个方法用于在 Namenode HA 切换时将当前 `DatanodeStorageInfo` 设置为 `stale` 状态。

```

void markStaleAfterFailover() {
    heartbeatedSinceFailover = false;
    blockContentsStale = true;
}

```

`DatanodeStorageInfo` 中的其他方法的实现都比较简单，大多是对定义字段的 `get/set` 方法，这里不再赘述。

3.3.3 DatanodeManager

了解了 `DatanodeDescriptor` 以及 `DatanodeStorageInfo` 类的实现后，我们来介绍 Namenode 对 `Datanode` 的管理类 `DatanodeManager`。本节首先会介绍 `DatanodeManager` 中比较重要的几个属性，然后重点介绍 `DatanodeManager` 中 `Datanode` 注册、撤销以及心跳流程的实现。

1. DatanodeManager 字段

`DatanodeManager` 类中记录了在 Namenode 上注册的 `Datanode`，以及这些 `Datanode` 在网络中的拓扑结构等信息。`DatanodeManager` 定义的字段如图 3-48 所示。

© DatanodeManager	
LOG	Log
namesystem	Namesystem
blockManager	BlockManager
heartbeatManager	HeartbeatManager
decommissionthread	Daemon
datanodeMap	NavigableMap<String, DatanodeDescriptor>
networkTopology	NetworkTopology
host2DatanodeMap	Host2NodesMap
dnsToSwitchMapping	DNSToSwitchMapping

图 3-48 DatanodeManager 字段

- `datanodeMap`: 维护 `StorageId -> DatanodeDescriptor` 的映射关系。
- `host2DatanodeMap`: 维护 `host -> DatanodeDescriptor` 的映射关系。这里为什么要维护两个 `Datanode` 的映射关系呢？这是因为 `Datanode` 的 `storageId` 是有可能发生变化的。一般情况下这两个映射关系是一致的，但是可能会出现 `Datanode` 重新启动后使用一个新的 `storageId` 注册的情况，这时候就需要对 `datanodeMap` 字段进行更新。这种情况会在后面的 `registerDatanode()` 方法介绍中具体说明。
- `networktopology`: 维护整个网络的拓扑结构。
- `decommissionthread`: 一个线程类周期性地调用 `checkDecommissionState()` 方法设置 `DatanodeDescriptor` 的 `decommission` 状态。

2. 添加和撤销 Datanode

HDFS 的一个重要特征就是具有弹性，也就是当 HDFS 需要增加容量时，可以动态地向集群中添加新的 `Datanode`；同理，当 HDFS 需要减小规模时，可以动态地撤销已经存在的 `Datanode`。无论是添加还是撤销 `Datanode` 的操作，都不会影响 HDFS 服务。

HDFS 提供了 `dfs.hosts` 文件（由配置项 `dfs.hosts` 指定，又称 `include` 文件）以及 `dfs.hosts.exclude` 文件（由配置项 `dfs.hosts.exclude` 指定，简称 `exclude` 文件）管理接入到 HDFS 的 `Datanode`。`include` 文件指定了可以连接到 `Namenode` 的 `Datanode` 列表，`exclude` 文件指定了不能连接到 `Namenode` 的 `Datanode` 列表，这两个文件都是文本文件，一行表示一个 `Datanode`。一个未定义或空的 `include` 文件意味着所有 `Datanode` 都可以连接到 `Namenode`。`Namenode` 会按照表 3-3 所示的规则决定所采取的动作。

表 3-3 Include & exclude 文件示意

在 include 文件中	在 exclude 文件中	示意
No	No	节点不可以连接
No	Yes	节点不可以连接
Yes	No	节点可以连接
Yes	Yes	节点可以连接，但是要撤销

HDFS 管理员将一个 `Datanode` 添加到集群中时，需要在 `include` 文件中添加一条该 `Datanode` 的记录，然后调用“`dfsadmin -refreshNodes`”命令刷新名字节点信息，最后才能启动 `Datanode`。同理，撤销节点则通过 `exclude` 文件，管理员将要撤销的节点信息添加到 `exclude` 文件中，也是调用“`dfsadmin -refreshNodes`”命令，`Namenode` 就会开始撤销节点操作。被撤销节点上的数据块会被复制到集群中的其他 `Datanode` 上，在这个过程中 `Datanode` 处于“正在撤销状态”（请参考 `DatanodeInfo.AdminState`），数据复制完成后 `Datanode` 状态会转变为“已撤销”，这时就可以关闭 `Datanode` 了。

(1) refreshNodes()

执行“`dfsadmin -refreshNodes`”命令会调用 RPC 方法 `ClientProtocol.refreshNodes()` 通知 `Namenode` 更新 `include` 文件和 `exclude` 文件，这个操作最终会由 `DatanodeManager.refreshNodes()`

方法响应。`refreshNodes()`方法会首先调用 `refreshHostsReader()`方法将 `include` 文件与 `exclude` 文件加载到 `hostFileManager` 对象中，之后调用 `refreshDatanodes()`刷新所有的数据节点。

```
public void refreshNodes(final Configuration conf) throws IOException {
    refreshHostsReader(conf);          // 加载 include 文件与 exclude 文件至 hostFileManager
    namesystem.writeLock();
    try {
        refreshDatanodes();           // 刷新所有的数据节点
        // ...
    } finally {
        namesystem.writeUnlock();
    }
}

/**加载 include/exclude 文件*/
private void refreshHostsReader(Configuration conf) throws IOException {
    if (conf == null) {
        conf = new HdfsConfiguration();
    }
    // 将更新的 include/exclude 文件加载至 hostFileManager 对象中保存
    this.hostFileManager.refresh(conf.get(DFSConfigKeys.DFS_HOSTS, ""),
        conf.get(DFSConfigKeys.DFS_HOSTS_EXCLUDE, ""));
}
```

`refreshDatanodes()` 方法会遍历 `DatanodeManager.datanodeMap` 字段中保存的所有 `DatanodeDescriptor` 对象，并按照表 3-3 中的规则进行处理。对于不可以连接到 Namenode 的 `Datanode`，设置对应的 `DatanodeDescriptor.isAllowed` 字段为 `false`，表明当前 `Datanode` 不可以接入 HDFS 集群。对于 `exclude` 文件中的节点，需要进行撤销操作，则调用 `startDecommission()` 开始撤销操作；不在 `exclude` 文件中的节点，则调用 `stopDecommission()` 停止撤销操作。

```
private void refreshDatanodes() {
    for(DatanodeDescriptor node : datanodeMap.values()) {
        // 不在 include 文件中
        if (!hostFileManager.isIncluded(node)) {
            // 直接设置为已撤销状态，DatanodeDescriptor.isAllowed=false
            // 不会拷贝 Datanode 上的数据块
            // 所以，在撤销节点时，先在 exclude 文件中添加，撤销结束后再从 include 文件中删除
            node.setDisallowed(true);
        } else {
            if (hostFileManager.isExcluded(node)) {
                // 在 exclude 文件中，则直接调用 startDecommission() 开始撤销操作
                startDecommission(node);
            } else {
                // 不在 exclude 文件中，则停止撤销操作
                stopDecommission(node);
            }
        }
    }
}
```

我们在 `DatanodeDescriptor` 小节中已经介绍过，`DatanodeInfo.adminState` 字段用于标识当前 `Datanode` 的状态。如果 `adminState==null`，则表明当前 `Datanode` 处于正常状态；如果为 `AdminStates.DECOMMISSION_INPROGRESS`，则表明 `Datanode` 处于正在撤销状态；如果为 `AdminStates.DECOMMISSIONED`，则表明 `Datanode` 已经撤销了。`startDecommission()`方法会首先调用 `heartbeatManager.startDecommission()`将当前 `Datanode` 对应的 `DatanodeDescriptor.adminState` 设置为 `AdminStates.DECOMMISSION_INPROGRESS` 状态。之后调用 `checkDecommissionState()` 检查撤销操作是否完成，如果完成则将 `DatanodeDescriptor.adminState` 设置为 `AdminStates.DECOMMISSIONED`。`checkDecommissionState()`方法会调用 `BlockManager.isReplicationInProgress()`判断当前节点上保存的所有数据块是否满足副本系数，如果不满足则将该数据块加入 `neededReplications` 中进行复制操作，撤销节点上数据块的复制操作都是由这个方法触发的。

```
private void startDecommission(DatanodeDescriptor node) {
    // 判断当前节点不为撤销相关状态
    if (!node.isDecommissionInProgress() && !node.isDecommissioned()) {
        // 设置 DatanodeDescriptor.adminState 状态
        heartbeatManager.startDecommission(node);
        node.decommissioningStatus.setStartTime(now());
        // 检查当前的撤销操作是否完成
        checkDecommissionState(node);
    }
}
```

`stopDecommission()`方法的逻辑也是类似的，首先调用 `heartbeatManager.stopDecommission()`方法将 `DatanodeDescriptor.adminState` 设置为 `null`，之后调用 `blockManager.processOverReplicatedBlocksOnReCommission()`将由于节点重新上架而造成超出副本系数的数据块删除。

```
void stopDecommission(DatanodeDescriptor node) {
    if (node.isDecommissionInProgress() || node.isDecommissioned()) {
        // 设置 DatanodeDescriptor.adminState 状态
        heartbeatManager.stopDecommission(node);
        if (node.isAlive) {
            // 由于节点重新上架，需要对超出备份数目的数据块进行判断，并进行删除操作
            blockManager.processOverReplicatedBlocksOnReCommission(node);
        }
    }
}
```

(2) checkDecommissionState()

`refreshNodes()`操作执行完成后，被撤销节点往往正在进行数据块备份操作，也就是处于正在撤销状态（`AdminStates.DECOMMISSION_INPROGRESS`）。当撤销节点完成数据块备份操作后，由什么方法将节点状态从正在撤销状态转换为已撤销状态呢？`DatanodeManager` 中持有一个 `decommissionthread` 线程，这个线程定期执行 `DecommissionManager.Monitor` 类的 `run()` 方法，`run()` 方法会调用 `check()` 方法执行检查节点状态的操作，`check()` 方法会调用 `DatanodeManager.checkDecommissionState()`方法检查撤销操作是否完成，如果完成了则将节点

状态设置为“已撤销”状态。`check()`方法的代码如下:

```
private void check() {
    final DatanodeManager dm = blockmanager.getDatanodeManager();
    int count = 0;
    // 循环检查所有的 Datanode
    for(Map.Entry<String, DatanodeDescriptor> entry
        : dm.getDatanodeCyclicIteration(firstkey)) {
        final DatanodeDescriptor d = entry.getValue();
        firstkey = entry.getKey();
        if (d.isDecommissionInProgress()) {
            try {
                // DatanodeManager.checkDecommissionState() 方法检查撤销操作是否完成
                dm.checkDecommissionState(d);
            } catch(Exception e) {
                // ...
            }
        }
    }
}
```

`checkDecommissionState()` 方法我们在上一小节中已经简单介绍过了, 它调用 `blockManager.isReplicationInProgress()` 方法判断撤销操作是否完成, 如果完成了撤销操作, 则设置节点状态为“已撤销”状态。

```
boolean checkDecommissionState(DatanodeDescriptor node) {
    if (node.isDecommissionInProgress()) {
        if (!blockManager.isReplicationInProgress(node)) {
            node.setDecommissioned();
        }
    }
    return node.isDecommissioned();
}
```

`blockManager.isReplicationInProgress()` 方法用于判断当前 `Datanode` 上的所有数据块是否有足够的备份数量, 如果所有数据块的副本数量都满足配置, 那么节点就成功地完成了撤销操作。`isReplicationInProgress()` 会遍历当前 `Datanode` 上保存的所有数据块, 然后调用 `countNodes()` 方法计算当前数据块的状态, 对于副本数量小于期望副本数量的数据块, 则加入 `neededReplications` 队列中进行复制。

```
boolean isReplicationInProgress(DatanodeDescriptor srcNode) {
    boolean status = false;
    boolean firstReplicationLog = true;
    int underReplicatedBlocks = 0; // 处于复制状态的数据块数量 (复制没结束)
    int decommissionOnlyReplicas = 0; // 只存在于撤销节点上的数据块数量
    int underReplicatedInOpenFiles = 0; // 正在写数据的数据块数量
    final Iterator<? extends Block> it = srcNode.getBlockIterator();
    while(it.hasNext()) { // 遍历当前 Datanode 保存的所有数据块
        final Block block = it.next();
```

Hadoop 2.X HDFS 源码剖析

```
BlockCollection bc = blocksMap.getBlockCollection(block);

if (bc != null) {
    NumberReplicas num = countNodes(block); // 获取当前数据块的状态
    int curReplicas = num.liveReplicas(); // 数据块有效的副本数量
    int curExpectedReplicas = getReplication(block); // 数据块期望的副本数量

    if (isNeededReplication(block, curExpectedReplicas, curReplicas)) {
        if (curExpectedReplicas > curReplicas) {
            if (bc.isUnderConstruction()) {
                if (block.equals(bc.getLastBlock()) && curReplicas > minReplication) {
                    continue;
                }
                underReplicatedInOpenFiles++; // 正在进行写操作的副本数量加 1
            }

            if (!status) {
                status = true;
                if (firstReplicationLog) {
                    logBlockReplicationInfo(block, srcNode, num);
                }
                if (curReplicas >= defaultReplication) {
                    status = false;
                    firstReplicationLog = false;
                }
            }
            underReplicatedBlocks++; // 处于复制状态的副本数量加 1
            if ((curReplicas == 0) && (num.decommissionedReplicas() > 0)) {
                decommissionOnlyReplicas++; // 只存在于当前撤销节点上的副本数量加 1
            }
        }
    }
    if (!neededReplications.contains(block) &&
        pendingReplications.getNumReplicas(block) == 0 &&
        namesystem.isPopulatingReplQueues()) {
        // 将小于期望副本数量的节点都加入 neededReplications 中进行复制
        neededReplications.add(block,
                                curReplicas,
                                num.decommissionedReplicas(),
                                curExpectedReplicas);
    }
}

// 记录撤销状态
srcNode.decommissioningStatus.set(underReplicatedBlocks,
    decommissionOnlyReplicas,
    underReplicatedInOpenFiles);
return status;
}
```

需要注意, 如果 `startDecommission()` 调用 `isReplicationInProgress()` 时节点刚进入撤销状态, 这时 `countNodes()` 方法并不将撤销节点上的副本计入有效副本中 (`liveReplicas`), 所以有效副本数量总是小于期望副本数量, 在这种情况下, `isReplicationInProgress()` 会将撤销节点上保存的数据块加入 `neededReplications` 队列中, 并执行复制操作。另一种情况是, 当数据节点被撤销后, 节点上的某个数据块完成了写操作并通过 RPC 接口汇报给 Namenode 时, Namenode 会将这个副本加入当前 Datanode 保存的数据块链表中, 当 `DatanodeManager.decommissionthread` 调用 `isReplicationInProgress()` 方法触发检查操作时, 会将新写入的这个数据块加入 `neededReplications` 队列中。可以看到, HDFS 的撤销流程保证了撤销节点上的数据块副本不会因为节点撤销而丢失。

3. Datanode 的启动

Datanode 启动时, 需要与 Namenode 进行握手、注册和数据块上报三个操作。这三个操作分别对应于 DatanodeProtocol 的 `versionRequest()`、`registerDatanode()` 以及 `blockReport()` 方法。

(1) 握手操作

握手请求由 `NameNodeRpcServer` 实现, 非常简单, 它直接返回命名空间的信息, 如下代码所示。

```
public NamespaceInfo versionRequest() throws IOException {
    namesystem.checkSuperuserPrivilege();
    return namesystem.getNamespaceInfo();
}
```

(2) 注册

Datanode 的注册请求会由 `FSNamesystem.registerDatanode()` 方法响应。Datanode 会为注册的 Datanode 分配唯一的 `storageId` 作为标识 (`storageId` 在 `datanodeMap` 中作为 key, 用于获取 `DatanodeDescriptor` 对象)。需要注意的是, 数据节点可以重复发送注册信息, 并且 Datanode 的 `storageId` 是有可能发生改变的 (当数据节点上的数据都被擦除时)。同时在 Datanode 注册时, `DatanodeManager` 还需要对 Datanode 是否可以接入进行判断 (`include/exclude` 文件判断)。

Datanode 的注册情况可以分为以下三种。

- 该 Datanode 没有注册过。
- 该 Datanode 注册过, 但是这次注册使用了新的 `storageId`, 表明该数据节点的存储空间已经被清理过, 原有的数据块副本都被删除了。
- 该 Datanode 注册过, 这次是重复注册。

下面我们看一下 `FSNamesystem.registerDatanode()` 方法的实现。由于 `registerDatanode()` 的方法体比较大, 我们将这个方法分为若干个部分进行讲解。在第一小节介绍中我们知道 `DatanodeManager` 中维护着两个 `DatanodeDescriptor` 对象的映射关系, 这里用 `nodeS` 表明从 `DatanodeManager.datanodeMap` 字段中通过 `StorageId` 获得的 `DatanodeDescriptor` 对象, 用 `nodeN` 表明从 `DatanodeManager.host2DatanodeMap` 字段中通过 `hostname` 获得的 `DatanodeDescriptor`

对象。

```
// 判断是否在 include 中
if (!hostFileManager.isIncluded(nodeReg)) {
    throw new DisallowedDatanodeException(nodeReg);
}

DatanodeDescriptor nodeS = getDatanode(nodeReg.getDatanodeUuid());
DatanodeDescriptor nodeN = host2DatanodeMap.getDatanodeByXferAddr(
    nodeReg.getIpAddr(), nodeReg.getXferPort());
```

那么对应于上述三种情况：

- **nodeS == null && nodeN == null**，对应于 Datanode 第一次注册的情况。将新注册的 Datanode 添加到 datanodeMap 以及 host2DatanodeMap 中，然后更新网络拓扑，检查节点是否撤销等。

```
DatanodeDescriptor nodeDescr
    = new DatanodeDescriptor(nodeReg, NetworkTopology.DEFAULT_RACK);
boolean success = false;
try {
    // 更新网络拓扑
    nodeDescr.setNetworkLocation(resolveNetworkLocation(nodeDescr));
    networktopology.add(nodeDescr);
    nodeDescr.setSoftwareVersion(nodeReg.getSoftwareVersion());

    // 添加到 datanodeMap 以及 host2DatanodeMap 中
    addDatanode(nodeDescr);
    // 检查是否撤销
    checkDecommissioning(nodeDescr);

    // 在 heartbeatManager 中添加该数据节点
    heartbeatManager.addDatanode(nodeDescr);
    success = true;
    incrementVersionCount(nodeReg.getSoftwareVersion());
} finally {
    //...
}
```

- **nodeN != null && nodeN != nodeS**，对应于已经注册的 Datanode 使用新的 storageId 注册的情况。也就是原先在 Datanode 上保存的数据块都无效了，需要清理 Namenode 中这个 Datanode 的信息，后续的处理则与第一种情况相似。

```
if (nodeN != null && nodeN != nodeS) {
    // 删除 DatanodeDescriptor 对象
    removeDatanode(nodeN);
    // 从 datanodeMap 以及 host2DatanodeMap 中删除
    wipeDatanode(nodeN);
    nodeN = null;
}
```

这里调用了两个删除方法，即 `removeDatanode(nodeN)` 和 `wipeDatanode(nodeN)`。其中 `removeDatanode()` 方法删除了 Namenode 内存中所有该 Datanode 对应的 `DatanodeDescriptor` 对象，同时从 `BlockManager.blocksMap` 中删除该 Datanode 存储的所有数据块；而 `wipeDatanode()` 则将 `DatanodeManager` 内部的两个数据结构 `datanodeMap` 和 `host2DatanodeMap` 上的 `DatanodeDescriptor` 对象删除。`removeDatanode()` 和 `wipeDatanode()` 方法的实现请参考以下代码。

```
private void wipeDatanode(final DatanodeID node) {
    final String key = node.getStorageID();
    synchronized (datanodeMap) {
        host2DatanodeMap.remove(datanodeMap.remove(key));
    }
}

private void removeDatanode(DatanodeDescriptor nodeInfo) {
    assert namesystem.hasWriteLock();
    heartbeatManager.removeDatanode(nodeInfo);
    blockManager.removeBlocksAssociatedTo(nodeInfo);
    networkTopology.remove(nodeInfo);
    decrementVersionCount(nodeInfo.getSoftwareVersion());
    namesystem.checkSafeMode();
}
```

■ `nodeS != null`，对应于第三种情况，也就是 Datanode 刷新注册。这时只需用刷新注册的信息更新 Namenode 内存中保存的 Datanode 原有信息即可。

```
// 更新网络拓扑、节点等信息
getNetworkTopology().remove(nodeS);
if (shouldCountVersion(nodeS)) {
    decrementVersionCount(nodeS.getSoftwareVersion());
}
nodeS.updateRegInfo(nodeReg);

nodeS.setSoftwareVersion(nodeReg.getSoftwareVersion());
nodeS.setDisallowed(false);

nodeS.setNetworkLocation(resolveNetworkLocation(nodeS));
getNetworkTopology().add(nodeS);

heartbeatManager.register(nodeS);
incrementVersionCount(nodeS.getSoftwareVersion());
checkDecommissioning(nodeS);
```

（3）数据块上报

Datanode 成功完成握手和注册操作后，就会进行数据块上报操作，将当前 Datanode 上存储的数据块上报给 Namenode。这个方法最终会由 `BlockManager.processReport()` 方法响应，如果是第一次块汇报，则调用 `processFirstBlockReport()` 方法，因为效率会高很多；否则调用私有的 `processReport()` 方法。当进行过一次块汇报时，我们还需要判断这个节点之前是否为 `stale`

状态,如果是,则需要对 `postponedMisreplicatedBlock` 进行重新扫描,删除那些已经不是 `stale` 状态的数据块。

```
public boolean processReport(final DatanodeID nodeID,
    final DatanodeStorage storage,
    final BlockListAsLongs newReport) throws IOException {
    // ...

    if (node.numBlocks() == 0) {
        // 对于第一次块汇报,调用 processFirstBlockReport()
        processFirstBlockReport(node, newReport);
    } else {
        // 不是第一次块汇报,则调用私有的 processReport() 方法
        processReport(node, newReport);
    }

    //...

    boolean staleBefore = node.areBlockContentsStale();
    node.receivedBlockReport();
    // 如果节点之前是 stale 状态,由于发起了新的 BlockReport,则该节点脱离 stale 状态
    if (staleBefore && !node.areBlockContentsStale()) {
        rescanPostponedMisreplicatedBlocks();
    }
    // ...
}
```

数据块上报操作我们在 `BlockManager` 类的块汇报小节中已经介绍过了,请读者参考。

4. Datanode 的心跳

通过 `Datanode` 章 `BlockPoolManager` 节的 `BPServiceActor` 小节的学习我们知道, `Datanode` 的 `BPServiceActor` 对象会以 `dnConf.heartBeatInterval` (默认配置是 3 秒) 间隔向 `Namenode` 发送心跳。`Datanode` 向 `Namenode` 发送的心跳信息包括: `Datanode` 的注册信息、`Datanode` 的存储信息 (使用容量、剩余容量等)、缓存信息、当前 `Datanode` 写文件的连接数、读写数据使用的线程数等描述当前 `Datanode` 负载的信息。

`Namenode` 收到 `Datanode` 的心跳之后,会调用 `BlockManager.handleHeartBeat()` 方法返回一个心跳响应 `HeartbeatResponse`。这个心跳响应中包含一个 `DatanodeCommand` 的数组,用来携带 `Namenode` 对 `Datanode` 的指令,例如数据块副本的复制、删除、缓存等指令。

`NameNode` 还会启动一个线程,负责周期性地检测所有 `Datanode` 上报心跳的情况,对于长时间没有上报心跳的 `Datanode`,则认为该 `Datanode` 出现故障不能正常工作,这时会调用 `DatanodeManager.removeDatanode()` 删除该数据节点。

所以对于 `Datanode` 的心跳处理,我们分为两个部分进行介绍:心跳信息处理以及心跳检查。

(1) 心跳信息处理

Datanode 发送的心跳请求将会由 `FSNamesystem.handleHeartbeat()` 方法处理, 这个方法会首先调用 `DatanodeManager.handleHeartbeat()` 对心跳信息进行处理, 并返回 Namenode 对当前 Datanode 下达的指令列表。之后构造一个 `NNHASStatusHeartbeat` 对象描述当前 Namenode 的 HA 状态。最后将生成的指令列表与 `NNHASStatusHeartbeat` 对象封装到 `HeartbeatResponse` 中, 返回给 Datanode。

这里我们重点了解 `DatanodeManager.handleHeartbeat()` 部分的代码, 可以按照逻辑将代码分为三个部分。

- 对发送心跳的数据节点进行检查: 检查该数据节点能否连接到 Namenode (例如在 `exclude` 文件中), 如果不能连接则抛出异常。检查该节点是否在 Namenode 上注册过, 如果没有注册, 则向该 Datanode 发送 `DNA_REGISTER` 注册指令。Datanode 接收到指令后会重新向 Namenode 发起注册请求。

```
public DatanodeCommand[] handleHeartbeat(DatanodeRegistration nodeReg,
    StorageReport[] reports, final String blockPoolId,
    long cacheCapacity, long cacheUsed, int xceiverCount,
    int maxTransfers, int failedVolumes
    ) throws IOException {
    synchronized (heartbeatManager) {
        synchronized (datanodeMap) {
            DatanodeDescriptor nodeinfo = null;
            try {
                nodeinfo = getDatanode(nodeReg);
            } catch (UnregisteredNodeException e) {
                return new DatanodeCommand[] {RegisterCommand.REGISTER};
            }

            // 第一部分代码, 检查当前 Datanode 能否连接到 Namenode
            // 在 refreshNodes() 小节中介绍过, DatanodeInfo.disallowed == true 表明节点不可连接
            if (nodeinfo != null && nodeinfo.isDisallowed()) {
                setDatanodeDead(nodeinfo);
                throw new DisallowedDatanodeException(nodeinfo);
            }

            // 确认当前 Datanode 已经在 Namenode 上注册, 否则发送注册指令
            if (nodeinfo == null || !nodeinfo.isAlive()) {
                return new DatanodeCommand[] {RegisterCommand.REGISTER};
            }

            // 第二部分、第三部分代码
            // ...
        }
    }
}
```

- Namenode 从 Datanode 的心跳中取出负载信息, 调用 `HeartbeatManager.updateHeartbeat()` 方法更新整个集群的负载信息, 同时也更新了节点的心跳事件。

```
// 第二部分代码, 调用 HeartbeatManager.updateHeartbeat() 更新集群负载信息
heartbeatManager.updateHeartbeat(nodeinfo, reports,
                                cacheCapacity, cacheUsed,
                                xceiverCount, failedVolumes);
```

HeartbeatManager.updateHeartbeat()方法会调用 DatanodeDescriptor.updateHeartbeat()方法更新 DatanodeDescriptor 中保存的节点信息。这个方法还执行了一个非常重要的操作, 就是根据这次心跳信息找出 Datanode 上的所有故障存储, 并将故障存储对应的 DatanodeStorageInfo 的 state 字段设置为 FAILED, 为之后 HeartbeatManager 的心跳处理做准备。

- Namenode 为 Datanode 生成名字节点指令, 如果当前名字节点还处于安全模式中, 则返回空指令; 否则依次生成数据块恢复指令 (DNA_RECOVERBLOCK)、数据块复制指令 (DNA_TRANSFER)、数据块删除指令 (DNA_INVALIDATE)、缓存相关指令 (DNA_CACHE&DNA_UNCACHE)、balancer 带宽指令 (DNA_BALANCERBANDWIDTHUPDATE)。最后将指令返回给 Datanode。

```
// 第三部分代码, 生成给 Datanode 的指令
// 如果当前 Namenode 处于安全模式中, 则不返回任何指令
if(namesystem.isInSafeMode()) {
    return new DatanodeCommand[0];
}

// 生成数据块恢复指令
BlockInfoUnderConstruction[] blocks = nodeinfo
    .getLeaseRecoveryCommand(Integer.MAX_VALUE);
if (blocks != null) {
    BlockRecoveryCommand brCommand = new BlockRecoveryCommand(
        blocks.length);
    // ...具体内容请参考后面的详细分析
}
return new DatanodeCommand[] { brCommand };

final List<DatanodeCommand> cmds = new ArrayList<DatanodeCommand>();
// 生成复制指令
List<BlockTargetPair> pendingList = nodeinfo.getReplicationCommand(
    maxTransfers);
if (pendingList != null) {
    cmds.add(new BlockCommand(DatanodeProtocol.DNA_TRANSFER, blockPoolId,
        pendingList));
}
// 生成删除指令
Block[] blks = nodeinfo.getInvalidateBlocks(blockInvalidateLimit);
if (blks != null) {
    cmds.add(new BlockCommand(DatanodeProtocol.DNA_INVALIDATE,
        blockPoolId, blks));
}
// 生成缓存指令
```

```

boolean sendingCachingCommands = false;
long nowMs = Time.monotonicNow();
if (shouldSendCachingCommands &&
    ((nowMs - nodeinfo.getLastCachingDirectiveSentTimeMs()) >=
     timeBetweenResendingCachingDirectivesMs)) {
    // 向缓存中添加数据块
    DatanodeCommand pendingCacheCommand =
        getCacheCommand(nodeinfo.getPendingCached(), nodeinfo,
            DatanodeProtocol.DNA_CACHE, blockPoolId);
    if (pendingCacheCommand != null) {
        cmds.add(pendingCacheCommand);
        sendingCachingCommands = true;
    }
    // 从缓存中删除数据块
    DatanodeCommand pendingUncacheCommand =
        getCacheCommand(nodeinfo.getPendingUncached(), nodeinfo,
            DatanodeProtocol.DNA_UNCACHE, blockPoolId);
    if (pendingUncacheCommand != null) {
        cmds.add(pendingUncacheCommand);
        sendingCachingCommands = true;
    }
}
blockManager.addKeyUpdateCommand(cmds, nodeinfo);
// 生成带宽指令
if (nodeinfo.getBalancerBandwidth() > 0) {
    cmds.add(new BalancerBandwidthCommand(nodeinfo.getBalancerBandwidth()));
    nodeinfo.setBalancerBandwidth(0);
}
// 返回所有指令
if (!cmds.isEmpty()) {
    return cmds.toArray(new DatanodeCommand[cmds.size()]);
}

```

(2) 名字节点指令生成

讲解清楚了心跳处理流程之后，我们来学习 Datanode 生成名字节点指令的细节。首先介绍数据块恢复操作的指令是如何生成的。当 Namenode 对一个数据块触发了租约恢复 (Lease Recover) 操作后，会在存储这个数据块的所有 Datanode 中选取一个 Datanode 作为主恢复节点 (Primary Datanode)，由这个节点协调所有参与租约恢复的 Datanode。Namenode 会调用主恢复 Datanode 对应的 DatanodeDescriptor 对象的 addBlockToBeRecovered() 方法，将需要恢复的数据块加入 DatanodeDescriptor.recoverBlocks 队列中。

handleHeartbeat() 方法会调用 getLeaseRecoveryCommand() 方法将 DatanodeDescriptor.recoverBlocks 队列中保存的所有需要进行块恢复操作的数据块都取出来，然后将所有超过 30s 未汇报心跳的节点从执行数据块恢复操作的 Datanode 队列中删除，最后构造数据块恢复指令并返回。

```

public DatanodeCommand[] handleHeartbeat( ... ) {

```

Hadoop 2.X HDFS 源码剖析

```
// ...

BlockInfoUnderConstruction[] blocks = nodeinfo
    .getLeaseRecoveryCommand(Integer.MAX_VALUE);
if (blocks != null) {
    BlockRecoveryCommand brCommand = new BlockRecoveryCommand(
        blocks.length);
    for (BlockInfoUnderConstruction b : blocks) {
        final DatanodeStorageInfo[] storages = b.getExpectedStorageLocations();
        // 忽略所有的 stale 状态的 Datanode (超过 30s 未汇报心跳)
        final List<DatanodeStorageInfo> recoveryLocations =
            new ArrayList<DatanodeStorageInfo>(storages.length);
        for (int i = 0; i < storages.length; i++) {
            if (!storages[i].getDatanodeDescriptor().isStale(staleInterval)) {
                recoveryLocations.add(storages[i]);
            }
        }
        // 筛选后有足够的数据节点, 那么在筛选后的节点上做恢复操作
        if (recoveryLocations.size() > 1) {
            brCommand.add(new RecoveringBlock(
                new ExtendedBlock(blockPoolId, b),
                DatanodeStorageInfo.toDatanodeInfos(recoveryLocations),
                b.getBlockRecoveryId()));
        } else {
            // 筛选后数据节点不够, 那么还在原有的数据节点上进行恢复操作
            brCommand.add(new RecoveringBlock(
                new ExtendedBlock(blockPoolId, b),
                DatanodeStorageInfo.toDatanodeInfos(storages),
                b.getBlockRecoveryId()));
        }
    }
}
return new DatanodeCommand[] { brCommand };
// ...
}
```

对于复制操作、删除操作以及缓存操作等指令, 只需从 `DatanodeDescriptor` 对应的 `replicateBlocks`、`invalidateBlocks`、`pendingCached` 以及 `pendingUncached` 队列中提取出数据块就好。这里我们以 `DatanodeDescriptor.getReplicationCommand()` 方法为例, `getReplicationCommand()` 从 `replicateBlocks` 队列中取出需要进行复制操作的 `BlockTargetPair` 列表并返回。

```
public List<BlockTargetPair> getReplicationCommand(int maxTransfers) {
    return replicateBlocks.poll(maxTransfers);
}
```

(3) 心跳检查——HeartbeatManager

`DatanodeManager` 中关于心跳处理的另一部分就是心跳检查, 由 `HeartbeatManager` 类实现。`HeartbeatManager.Monitor` 线程会定期调用 `HeartbeatManager.heartbeatCheck()` 方法检查所有数据节点是否更新了心跳并执行了清理操作 (`dfs.namenode.heartbeat.recheck-interval` 配置项配置检查间隔, 默认是 5 分钟)。

```

private class Monitor implements Runnable {
    private long lastHeartbeatCheck;
    private long lastBlockKeyUpdate;

    @Override
    public void run() {
        while(namesystem.isRunning()) {
            try {
                final long now = Time.now();
                if (lastHeartbeatCheck + heartbeatRecheckInterval < now) {
                    heartbeatCheck(); // 调用 heartbeatCheck() 检查心跳更新情况
                    lastHeartbeatCheck = now;
                }
                // ...
            } catch (Exception e) {
                // ...
            }
            try {
                Thread.sleep(5000); // 5秒
            } catch (InterruptedException ie) {
            }
        }
    }

    // ...
}

```

在 `heartbeatCheck()` 方法中, 如果发现 `Datanode` 在 `timeout` 的时间内还未上报心跳, 则认为 `Datanode` 发生故障(在 `isDatanodeDead()` 方法中实现)。对于故障的 `Datanode`, `heartbeatCheck()` 方法会调用 `DatanodeManager.removeDeadDatanode()` 方法从 `Namenode` 中删除这个数据节点的信息, 以及这个数据节点保存的所有数据块副本信息。`timeout` 是 `Datanode` 被认为故障的时长, 其计算公式如下:

$$\text{timeout} = 2 * \text{heartbeat.recheck.interval} + 10 * \text{dfs.heartbeat.interval}$$

同时, `heartbeatCheck()` 方法会查找出一个故障的 `Datanode` 存储, 这个故障存储所在的 `Datanode` 必须是正常的, 但是存储所对应的 `DatanodeStorageInfo` 对象的 `state` 状态为 `FAILED`。`Datanode` 存储的故障检测是在 `DatanodeDescriptor.updateHeartbeat()` 方法中执行的, 这部分内容请参考心跳信息处理小节中的介绍。对于故障的 `Datanode` 存储, `heartbeatCheck()` 方法会调用 `BlockManager.removeBlocksAssociatedTo()` 删除故障 `Datanode` 存储上的所有数据块副本信息。

`heartbeatCheck()` 方法的代码如下所示, 它首先进行故障 `Datanode` 以及故障存储的检测操作, 然后分别调用 `DatanodeManager.removeDeadDatanode` 和 `BlockManager.removeBlocksAssociatedTo()` 处理故障 `Datanode` 以及故障存储。

```

void heartbeatCheck() {

```

Hadoop 2.X HDFS 源码剖析

```
final DatanodeManager dm = blockManager.getDatanodeManager();
// 处于安全模式中，直接返回
if (namesystem.isInStartupSafeMode()) {
    return;
}

// 查找故障节点以及故障存储
boolean allAlive = false;
while (!allAlive) {
    // 保存找到的第一个故障节点
    DatanodeID dead = null;

    // 保存找到的第一个故障存储，注意故障存储所在的 Datanode 必须是正常运行的
    DatanodeStorageInfo failedStorage = null;

    // ...
    synchronized(this) {
        for (DatanodeDescriptor d : datanodes) { // 遍历所有 Datanode
            // 调用 isDatanodeDead() 判断当前节点是否故障
            if (dead == null && dm.isDatanodeDead(d)) {
                stats.incrExpiredHeartbeats();
                dead = d; // 保存找到的第一个故障节点
            }

            DatanodeStorageInfo[] storageInfos = d.getStorageInfos();
            for (DatanodeStorageInfo storageInfo : storageInfos) { // 遍历节点上的所有存储
                if (failedStorage == null &&
                    storageInfo.areBlocksOnFailedStorage() &&
                    d != dead) {
                    failedStorage = storageInfo; // 保存找到的第一个故障存储
                }
            }
        }
    }
    // ...
}

allAlive = dead == null && failedStorage == null;
if (dead != null) {
    namesystem.writeLock();
    try {
        synchronized(this) {
            // 调用 DatanodeManager.removeDeadDatanode() 方法删除故障节点上的所有数据块信息
            dm.removeDeadDatanode(dead);
        }
    } finally {
        namesystem.writeUnlock();
    }
}
```

```

if (failedStorage != null) {
    namesystem.writeLock();
    try {
        if (namesystem.isInStartupSafeMode()) {
            return;
        }
        // 调用 BlockManager.removeBlocksAssociatedTo() 方法删除故障存储上的所有数据块信息
        synchronized(this) {
            blockManager.removeBlocksAssociatedTo(failedStorage);
        }
    } finally {
        namesystem.writeUnlock();
    }
}
}
}

```

3.4 租约管理

我们知道 HDFS 文件是 write-once-read-many，并且不支持客户端的并行写操作，那么这里就需要一种机制保证对 HDFS 文件的互斥操作。HDFS 提供了租约（Lease）机制来实现这个功能，租约是 HDFS 中一个很重要的概念，是 Namenode 给予租约持有者（LeaseHolder，一般是客户端）在规定时间内拥有文件权限（写文件）的合同。

在 HDFS 中，客户端写文件时需要先从租约管理器（LeaseManager）申请一个租约，成功申请租约之后客户端就成为了租约持有者，也就拥有了对该 HDFS 文件的独占权限，其他客户端在该租约有效时无法打开这个 HDFS 文件进行操作。Namenode 的租约管理器保存了 HDFS 文件与租约、租约与租约持有者的对应关系，租约管理器还会定期检查它维护的所有租约是否过期。租约管理器会强制收回过期的租约，所以租约持有者需要定期更新租约（renew），维护对该文件的独占锁定。当客户端完成了对文件的写操作，关闭文件时，必须在租约管理器中释放租约。

3.4.1 LeaseManager.Lease

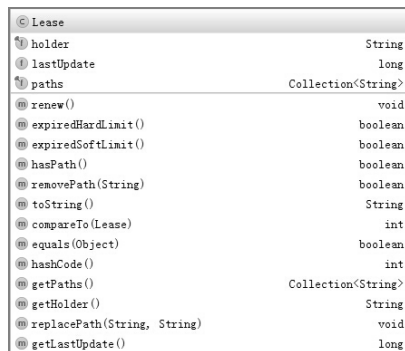
我们知道一个 HDFS 客户端是可以同时打开多个 HDFS 文件进行读写操作的，为了便于管理，在租约管理器中将一个客户端打开的所有文件组织在一起构成一条记录，也就是 LeaseManager.Lease 类。

图 3-49 给出了 Lease 类定义的所有字段，其中 holder 字段保存了客户端也就是租约持有者的信息，paths 字段保存了该客户端打开的所有 HDFS 文件的路径，lastUpdate 字段则保存了租约最后的更新时间。

Lease 类中还有三个比较重要的方法需要重点讲解，这些方法都是供 LeaseManager 管理

Lease 时调用的。

- **renew():** renew()方法用于更新客户端的 lastUpdate 最近更新时间。
- **expiredSoftLimit():** 用于判断当前租约是否超出了软限制 (softLimit)，软限制是写文件规定的租约超时时间，默认是 60 秒，不可以配置。
- **expiredHardLimit():** 用于判断当前租约是否超出了硬限制 (hardLimit)，硬限制是用于考虑文件关闭异常时，强制回收租约的时间，默认是 60 分钟，不可以配置。在 LeaseManager 中有一个内部类用于定期检查租约的更新情况，当超过硬限制时间时，会触发租约恢复机制。



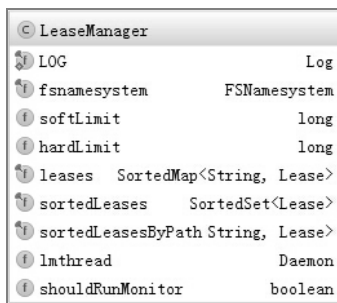
Lease	
holder	String
lastUpdate	long
paths	Collection<String>
renew()	void
expiredHardLimit()	boolean
expiredSoftLimit()	boolean
hasPath()	boolean
removePath(String)	boolean
toString()	String
compareTo(Lease)	int
equals(Object)	boolean
hashCode()	int
getPaths()	Collection<String>
getHolder()	String
replacePath(String, String)	void
getLastUpdate()	long

3.4.2 LeaseManager

图 3-49 Lease 类结构

LeaseManager 是 Namenode 中维护所有租约操作的类，它不仅仅保存了 HDFS 中所有租约的信息，提供租约的增、删、改、查方法，同时还维护了一个 Monitor 线程定期检查租约是否超时，对于长时间没有更新租约的文件（超过硬限制时间），LeaseManager 会触发租约恢复机制，然后关闭文件。

图 3-50 给出了 LeaseManager 定义的字段，在 LeaseManager 中使用数据结构 leases、sortedLeases 以及 sortedLeasesByPath 三个字段保存 Namenode 中的所有租约；使用 lmthread 字段保存租约检查线程；使用 softLimit 字段保存软限制时间（默认是 60 秒，不可以配置）；使用 hardLimit 字段保存硬限制时间（默认是 60 分钟，不可以配置）。



LeaseManager	
LOG	Log
fsnamesystem	FSNamesystem
softLimit	long
hardLimit	long
leases	SortedMap<String, Lease>
sortedLeases	SortedSet<Lease>
sortedLeasesByPath	String, Lease>
lmthread	Daemon
shouldRunMonitor	boolean

图 3-50 LeaseManager 字段

下面我们看一下 LeaseManager 保存租约的三个字段：leases、sortedLeases 和 sortedLeasesByPath。

- **leases:** 保存了租约持有者与租约的对应关系。
- **sortedLeases:** 以租约更新时间为顺序保存 LeaseManager 中的所有租约，如果更新时间相同，则按照租约持有者的字典序保存。

- **sortedLeasesByPath**: 保存了文件路径与租约的对应关系，以路径的字典序为顺序保存。

了解了 LeaseManager 定义的字段，下面我们学习 LeaseManager 提供的增加、删除、更新以及恢复租约的方法。

1. 添加租约——addLease()

当客户端创建文件和追加写文件时，FSNamesystem.startFileInternal()以及 appendFileInternal()方法都会调用 LeaseManager.addLease()为该客户端在 HDFS 文件上添加一个租约。addLease()方法有两个参数，其中 holder 参数保存租约的持有者信息，src 参数则保存创建或者追加写文件的路径，这两个参数分别对应于 ClientProtocol.create()或者 append()方法中的 clientName 和 src 参数。addLease()方法的实现也非常简单，先是通过 getLease()方法构造租约，然后在 LeaseManager 定义的保存租约的数据结构中添加这个租约的信息。

```
synchronized Lease addLease(String holder, String src) {
    Lease lease = getLease(holder);
    if (lease == null) {
        lease = new Lease(holder); // 构造 Lease 对象
        leases.put(holder, lease); // 在 LeaseManager.leases 字段中添加 Lease 对象
        sortedLeases.add(lease); // 在 LeaseManager.sortedLease 字段中添加 Lease 对象
    } else {
        renewLease(lease);
    }
    sortedLeasesByPath.put(src, lease); // 在 LeaseManager.sortedLeasesByPath 字段中添加 Lease 对象
    lease.paths.add(src);
    return lease;
}
```

addLease()在另外两种情况下也会被调用，就是 Namenode 读取 fsimage 文件时，fsimage 文件记录了当前 HDFS 文件处于构建状态中，这时需要重建这个构建中的文件并将文件对应的 INode 对象加入文件系统目录树中，然后还需要在 LeaseManager 中添加租约信息；以及在 Namenode 读取 editlog 时，editlog 记录了一个 OP_ADD 操作，也就是创建文件的操作，Namenode 创建完 INode 对象并添加到文件系统目录树之后，还需要在 LeaseManager 中添加租约信息。

2. 检查租约——FsNamesystem.checkLease()

如图 3-51 所示，当客户端已经成功打开了一个 HDFS 文件并添加租约后，客户端调用 abandonBlock()放弃新申请的数据块（客户端数据流管道建立失败时），调用 getAdditionalBlock()申请新的数据块（客户端完成了上一个数据块的写入，申请新的数据块时），调用 completeFileInternal()提交文件（客户端完成 HDFS 文件的写操作，提交文件时）等情况时，都需要检查租约是否正常。

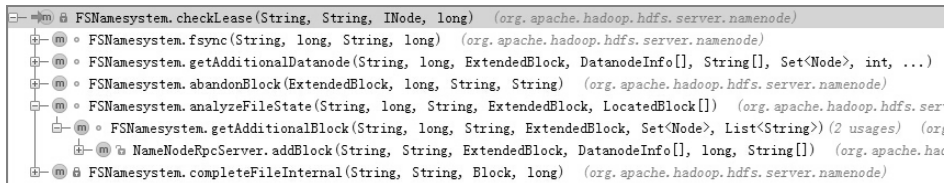


图 3-51 checkLease()调用关系

这里的检查操作是由 `FSNamesystem.checkLease()` 方法执行的，`checkLease()` 方法会检查当前 HDFS 文件是否存在、`INode` 是否是一个目录、文件是否处于构建中状态、文件是否已被删除、输入的文件租约持有者与实际的租约持有者是否相同。如果这些检查不正常，则抛出 `LeaseExpiredException`。

```
private INodeFile checkLease(String src, String holder, INode inode,
                             long fileId)
    throws LeaseExpiredException, FileNotFoundException {
    assert hasReadLock();
    final String ident = src + " (inode " + fileId + ")";
    if (inode == null) { // HDFS 文件不存在，则抛出异常
        Lease lease = leaseManager.getLease(holder);
        throw new LeaseExpiredException("...");
    }
    if (!inode.isFile()) { // INode 是一个目录，则抛出异常
        Lease lease = leaseManager.getLease(holder);
        throw new LeaseExpiredException("...");
    }
    final INodeFile file = inode.asFile();
    if (!file.isUnderConstruction()) { // 文件不处于构建中状态，则抛出异常
        Lease lease = leaseManager.getLease(holder);
        throw new LeaseExpiredException("...");
    }
    if (isFileDeleted(file)) { // 文件已经被删除，则抛出异常
        throw new FileNotFoundException(src);
    }
    String clientName = file.getFileUnderConstructionFeature().getClientName();
    if (holder != null && !clientName.equals(holder)) { // 租约信息与 LeaseManager 记录的不匹配，则抛出异常
        throw new LeaseExpiredException("...");
    }
    return file;
}
```

3. 租约更新——`renewLease()`

当客户端打开了一个文件用于写或者追加写操作时，`LeaseManager` 会保存这个客户端在该文件上的租约。客户端会启动一个 `LeaseRenewer` 定期更新租约，以防止租约过期。

租约更新操作是由 `FSNamesystem.renewLease()` 响应的，这个方法最终会调用

`LeaseManager.renewLease()`方法。`renewLease()`方法会首先从 `sortedLeases` 字段中移除这个租约，然后更新这个租约的最后更新时间，再重新加入 `sortedLeases` 中。这么做的原因是，`sortedLeases` 是一个以最后更新时间排序的集合，所以每次更新租约后，`sortedLeases` 中的顺序也需要重新改变。

```
synchronized void renewLease(Lease lease) {
    if (lease != null) {
        sortedLeases.remove(lease);
        lease.renew();
        sortedLeases.add(lease);
    }
}
```

4. 删除租约——`removeLease()`

`LeaseManager` 中的租约会在两种情况下被删除。

- Namenode 关闭构建中的 HDFS 文件时，会调用 `FSNamesystem.finalizeINodeFileUnderConstruction()` 方法将 `INode` 从构建状态转换成非构建状态，同时由于客户端已经完成了文件的写操作，所以需要从 `LeaseManager` 中删除该文件的租约，这里调用了 `removeLease()` 方法删除租约。
- 在进行目录树的删除操作时，对于已经打开的文件，如果客户端从文件系统目录树中移出该 HDFS 文件，则会调用 `removeLeaseWithPrefixPath()` 方法从 `LeaseManager` 中删除租约。

`removeLease()` 和 `removeLeaseWithPrefixPath()` 的实现都比较简单，从 `LeaseManager` 保存租约的数据结构中删除租约信息即可，读者可以直接参考以下代码：

```
synchronized void removeLeaseWithPrefixPath(String prefix) {
    for (Map.Entry<String, Lease> entry
         : findLeaseWithPrefixPath(prefix, sortedLeasesByPath).entrySet()) {
        removeLease(entry.getValue(), entry.getKey());
    }
}

synchronized void removeLease(String holder, String src) {
    Lease lease = getLease(holder);
    if (lease != null) {
        removeLease(lease, src);
    } else {
    }
}
```

5. 租约检查——Monitor 线程

我们知道租约管理器除了对租约提供增、删、改、查等操作外，还会定期检查所有租约，对于长时间没有进行租约更新的文件，`LeaseManager` 会对这个文件进行租约恢复操作，然后关闭这个文件。这里读者可能会有疑问，在什么情况下会出现租约过期呢？我们知道 HDFS

是一个分布式系统，客户端很有可能在打开一个文件之后出现故障，这也就造成了客户端不能完成租约更新以及写文件之后的租约删除操作，这时就会造成租约过期。

租约的定期检查操作是由 LeaseManager 的内部类 Monitor 执行的，Monitor 是一个线程类，它的 run() 方法会每隔 2 秒调用一次 LeaserManager.checkLeases() 方法检查租约。

在前面的小节中我们已经介绍过，LeaseManager 中有两个限制时间，其中软限制时间用于记录写文件规定的租约超时时间；硬限制时间则用于判断文件是否由于异常而未能正确关闭。在 checkLeases() 方法中就是使用硬限制时间（60 分钟）判断是否需要进行租约恢复操作的。

checkLeases() 方法会遍历 leaseManager 中管理的所有租约，找出所有超过硬限制时间而未更新的租约。由于租约保存了这个客户端打开的所有 HDFS 文件，所以 checkLeases() 方法会遍历这个租约上的所有文件，并调用 FSNamesystem.internalReleaseLease() 方法进行租约恢复操作。checkLeases() 方法的代码如下：

```
private synchronized boolean checkLeases() {
    boolean needSync = false;
    // 遍历 LeaseManager 中的所有租约
    for(; sortedLeases.size() > 0; ) {
        final Lease oldest = sortedLeases.first();
        // 如果当前租约没有超过硬限制时间，则直接返回，因为后面的租约并不需要判断
        if (!oldest.expiredHardLimit()) {
            return needSync;
        }

        // 租约超时情况处理
        final List<String> removing = new ArrayList<String>();

        // 遍历超时租约中的所有文件，对每一个文件进行租约恢复
        String[] leasePaths = new String[oldest.getPaths().size()];
        oldest.getPaths().toArray(leasePaths);
        for(String p : leasePaths) {
            try {
                // 调用 Fsnamesystem.internalReleaseLease() 对文件进行租约恢复
                boolean completed = fsnamesystem.internalReleaseLease(oldest, p,
                    HdfsServerConstants.NAMENODE_LEASE HOLDER);
                // 由于进行了恢复操作，需要在 editlog 中同步记录
                if (!needSync && !completed) {
                    needSync = true;
                }
            } catch (IOException e) {
                // 租约恢复出现异常，则加入 removing 队列中
                removing.add(p);
            }
        }
    }
}
```

```

// 租约恢复异常，则直接删除
for(String p : removing) {
    removeLease(oldest, p);
}
}
return needSync;
}

```

6. 租约恢复——Monitor 线程发起

对于 HDFS 文件的租约恢复操作是通过调用 `FSNamesystem.internalReleaseLease()` 实现的，这个方法用于将一个已经打开的文件进行租约恢复并关闭。如果成功关闭了文件，`internalReleaseLease()` 方法会返回 `true`；如果仅触发了租约恢复操作，则返回 `false`。我们知道租约恢复是针对已经打开的构建中的文件的，所以 `internalReleaseLease()` 会判断文件中所有数据块的状态，对于异常的状态则直接抛出异常。在 `checkLeases()` 方法中，对于调用 `FSNamesystem.internalReleaseLease()` 方法时抛出异常的租约，则直接调用 `removeLease()` 方法删除。

当文件处于构建状态时，有三种情况可以直接关闭文件，并返回 `true`。

- 这个文件所拥有的所有数据块都处于 `COMPLETED` 状态，也就是客户端还没有来得及关闭文件和释放租约就出现了故障，这时 `internalReleaseLease()` 可以直接调用 `finalizeINodeFileUnderConstruction()` 方法关闭文件并删除租约。
- 文件的最后一个数据块处于提交状态（`COMMITTED`），并且该数据块至少有一个有效的副本，这时可以直接调用 `finalizeINodeFileUnderConstruction()` 方法关闭文件并删除租约。
- 文件的最后一个数据块处于构建中状态，但这个数据块的长度为 0，且当前没有 `Datanode` 汇报接收了这个数据块，这种情况很可能是客户端向数据流管道中写数据前发生了故障，这时可以将最后一个未写入数据的数据块删除，之后调用 `finalizeINodeFileUnderConstruction()` 方法关闭文件并删除租约。

当最后一个数据块处于 `UNDER_RECOVERY` 或者 `UNDER_CONSTRUCTION` 状态，且这个数据块已经写入了数据时，则构造一个新的时间戳作为 `recoveryId`，调用 `initializeBlockRecovery()` 触发租约恢复，更新当前文件的租约持有者为“`HDFS_NameNode`”。`internalReleaseLease()` 的代码如下：

```

boolean internalReleaseLease(Lease lease, String src,
    String recoveryLeaseHolder) throws AlreadyBeingCreatedException,
    IOException, UnresolvedLinkException {

    final INodesInPath iip = dir.getLastINodeInPath(src);
    final INodeFile pendingFile = iip.getINode(0).asFile();
    int nrBlocks = pendingFile.numBlocks();    // 文件拥有的数据块数量
    int nrCompleteBlocks; // 文件拥有的数据块中处于 COMPLETE 状态的数量
    // ...

```

Hadoop 2.X HDFS 源码剖析

```
// 如果文件拥有的所有数据块都处于 COMPLETE 状态，则可以直接关闭文件，释放租约
if(nrCompleteBlocks == nrBlocks) {
    finalizeINodeFileUnderConstruction(src, pendingFile,
        iip.getLatestSnapshotId());
    return true; // closed!
}

// 在文件的所有数据块中，只有最后一个数据块以及倒数第二个数据块的状态可以不为
// COMPLETE，且状态只能是 COMMITTED
if(nrCompleteBlocks < nrBlocks - 2 ||
    nrCompleteBlocks == nrBlocks - 2 &&
    curBlock != null &&
    curBlock.getBlockUCState() != BlockUCState.COMMITTED) {
    // 如果最后一个数据块既不处于 COMMITTED 状态，也不处于 COMPLETE 状态，则抛出异常
    throw new IOException(message);
}

final BlockInfo lastBlock = pendingFile.getLastBlock();
BlockUCState lastBlockState = lastBlock.getBlockUCState();
BlockInfo penultimateBlock = pendingFile.getPenultimateBlock();
boolean penultimateBlockMinReplication = penultimateBlock == null ? true :
    blockManager.checkMinReplication(penultimateBlock);

switch(lastBlockState) {
case COMPLETE:
    // 不可能发生这种情况
    assert false : "Already checked that the last block is incomplete";
    break;
case COMMITTED:
    // 如果最后一个数据块处于 COMMITTED 状态，且数据块副本数量大于 1
    // 则关闭文件，释放租约
    if(penultimateBlockMinReplication &&
        blockManager.checkMinReplication(lastBlock)) {
        finalizeINodeFileUnderConstruction(src, pendingFile,
            iip.getLatestSnapshotId());
        return true;
    }
    // 由于最后一个数据块的副本数量不足，需要等待复制操作，则抛出异常
    // 无法执行租约的释放操作
    throw new AlreadyBeingCreatedException(message);
case UNDER_CONSTRUCTION:
case UNDER_RECOVERY:
    final BlockInfoUnderConstruction uc = (BlockInfoUnderConstruction)lastBlock;
    if (uc.getNumExpectedLocations() == 0) {
        uc.setExpectedLocations(blockManager.getStorages(lastBlock));
    }

    // 如果最后一个数据块未写入任何数据，则删除这个数据块，关闭文件
```

```

    if (uc.getNumExpectedLocations() == 0 && uc.getNumBytes() == 0) {
        pendingFile.removeLastBlock(lastBlock);
        finalizeINodeFileUnderConstruction(src, pendingFile,
            iip.getLatestSnapshotId());
        return true;
    }

    // 如果最后一个数据块写入了数据, 则调用 initializeBlockRecovery() 触发租约恢复
    // 并且更新租约的持有者为 HDFS_Namenode
    long blockRecoveryId = nextGenerationStamp(isLegacyBlock(uc));
    lease = reassignLease(lease, src, recoveryLeaseHolder, pendingFile);
    uc.initializeBlockRecovery(blockRecoveryId);
    leaseManager.renewLease(lease);

    // 租约恢复时不可以关闭文件, 需要等待租约恢复操作结束
    break;
}
return false;
}

```

下面我们看一下租约恢复的过程, 这里是在需要进行租约恢复的数据块上调用 `initializeBlockRecovery()` 方法, 该方法会遍历所有保存副本的数据节点, 选取一个最近一次进行汇报的数据节点作为主恢复节点, 然后向这个数据节点发送租约恢复指令, Namenode 会通过心跳将租约恢复的名字节点指令下发给该恢复节点。

```

public void initializeBlockRecovery(long recoveryId) {
    // 将数据块状态更改为 UNDER_RECOVERY
    setBlockUCState(BlockUCState.UNDER_RECOVERY);
    // 数据块恢复时间戳
    blockRecoveryId = recoveryId;

    // 遍历所有副本所在的 Datanode, 选取最近一次进行汇报的 Datanode 作为主恢复节点
    long mostRecentLastUpdate = 0;
    ReplicaUnderConstruction primary = null;
    primaryNodeIndex = -1;
    for(int i = 0; i < replicas.size(); i++) {
        if (!(replicas.get(i).isAlive() && !replicas.get(i).getChosenAsPrimary())) {
            continue;
        }
        final ReplicaUnderConstruction ruc = replicas.get(i);
        final long lastUpdate=ruc.getExpectedStorageLocation().getDatanodeDescriptor().
            getLastUpdate();
        if (lastUpdate > mostRecentLastUpdate) {
            primaryNodeIndex = i;
            primary = ruc;
            mostRecentLastUpdate = lastUpdate;
        }
    }
}

```

Hadoop 2.X HDFS 源码剖析

```
if (primary != null) {  
    // 向主恢复节点发送数据块恢复指令  
    primary.getExpectedStorageLocation().getDatanodeDescriptor().addBlock  
    ToBeRecovered(this);  
    primary.setChosenAsPrimary(true);  
}  
}
```

租约恢复指令是通过心跳响应携带给主恢复数据节点的，主恢复数据节点的租约恢复流程如图 3-52 所示。主恢复数据节点接收到指令后，会调用 `Datanode.recoverBlock()` 方法开始租约恢复，这个方法首先会通过 `InterDatanodeProtocol.initReplicaRecovery()` 方法向数据流管道中参与租约恢复的数据节点收集副本信息，副本信息会以 `ReplicaRecoveryInfo` 对象返回给主恢复节点。`initReplicaRecovery()` 方法会从该数据块的所有副本中选取一个最好的状态，作为所有副本恢复的目标状态。然后主恢复节点会调用 `InterDatanodeProtocol.updateReplicaUnderRecovery()` 方法同步所有 `Datanode` 上该数据块副本至目标状态。同步结束后，这些数据节点上的副本长度和时间戳将一致。最后，主恢复节点会调用 `DatanodeProtocol.commitBlockSynchronization()` 向名字节点报告这次租约恢复的结果。

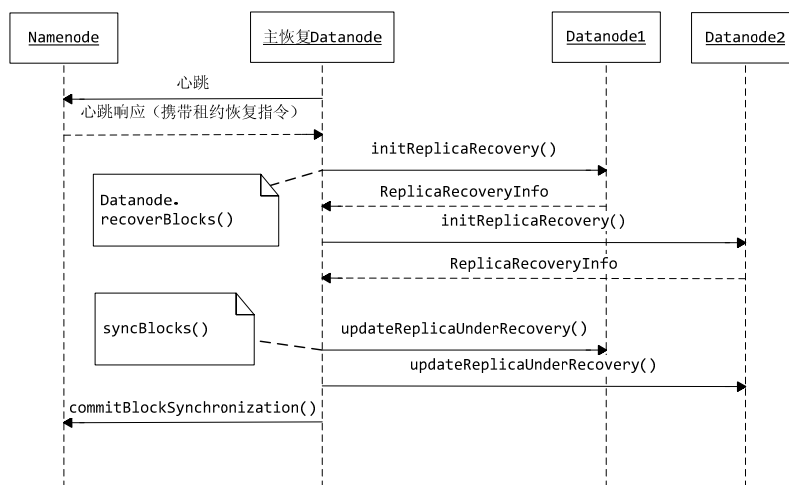


图 3-52 主恢复数据节点租约恢复流程图

上述流程中提到的 `InterDatanodeProtocol.initReplicaRecovery()` 以及 `updateReplicaUnderRecovery()` 方法会在第 4 章的 `FsDatasetImpl` 小节中介绍，这里不再重复介绍。现在我们看一下数据块同步的提交方法 `commitBlockSynchronization()`。`commitBlockSynchronization()` 方法用于将进行了租约恢复的数据节点上的副本信息与名字节点上的数据块信息同步。这个方法的参数也比较多，其中 `lastblock` 是被恢复的数据块，`newgenerationstamp` 是租约恢复以后新的时间戳，`newlength` 是租约恢复以后副本的长度，`closeFile` 用于指示是否关闭数据块对应的 HDFS

文件, `deleteblock` 用于指示是否直接删除这个数据块, `newtargets` 存储了租约恢复后保存这个数据块副本的数据节点列表, `newtargetstorages` 保存了数据节点的存储信息。

在数据块恢复过程中, 如果发现数据流管道中并不存在这个副本或者所有副本的长度都为 0, 则可以不用进行租约恢复操作, 直接删除这个数据块即可, 这时主恢复节点会将 `deleteblock` 字段设置为 `true`, `commitBlockSynchronization()` 方法会从 Namenode 中删除这个数据块, 然后关闭文件; 否则, `commitBlockSynchronization()` 方法进行数据块更新操作, 更新 Namenode 上数据块的时间戳和长度, 使 Namenode 上的数据块信息与 Datanode 上进行租约恢复后的副本一致。由于可能只有部分数据节点参与租约恢复, 所以还需要更新 `DatanodeStorageInfo` 上的数据块信息, 以及 `INodeFile` 中的数据块信息。

```
void commitBlockSynchronization(ExtendedBlock lastblock,
    long newgenerationstamp, long newlength,
    boolean closeFile, boolean deleteblock, DatanodeID[] newtargets,
    String[] newtargetstorages)
    throws IOException, UnresolvedLinkException {
    checkOperation(OperationCategory.WRITE);
    String src = "";
    waitForLoadingFSImage();
    writeLock();
    try {
        // 检查是否拥有命名空间的写权限
        // 检查是否处于安全模式

        // 检查提交的数据块是否存在, 如果不存在则抛出异常
        // 检查数据块对应的 HDFS 文件是否存在, 如果不存在则抛出异常
        // 检查这个 HDFS 文件是否已经被删除, 如果删除了则抛出异常
        // 如果文件不处于构建中状态, 或者最后一个数据块已经提交, 则抛出异常
        // 检查 recoveryId 是否和当前 Namenode 记录的 recoveryId 一致, 不一致则抛出异常

        // 如果设置了删除数据块, 则从 INode 以及 blocksMap 中删除数据块
        if (deleteblock) {
            Block blockToDel = ExtendedBlock.getLocalBlock(lastblock);
            boolean remove = iFile.removeLastBlock(blockToDel);
            if (remove) {
                blockManager.removeBlockFromMap(storedBlock);
            }
        }
        else {
            // 否则为更新数据块的情况
            // 将名字节点记录的数据块长度和时间戳与租约恢复后数据节点上的副本同步
            storedBlock.setGenerationStamp(newgenerationstamp);
            storedBlock.setNumBytes(newlength);

            // 查找出来租约恢复后存储当前数据块副本的 DatanodeDescriptor
            // 以及 DatanodeStorageInfo 对象
            ArrayList<DatanodeDescriptor> trimmedTargets =
```

```
        new ArrayList<DatanodeDescriptor>(newtargets.length);
ArrayList<String> trimmedStorages =
    new ArrayList<String>(newtargets.length);
if (newtargets.length > 0) {
    for (int i = 0; i < newtargets.length; ++i) {
        DatanodeDescriptor targetNode =
            blockManager.getDatanodeManager().getDatanode(newtargets[i]);
        if (targetNode != null) {
            trimmedTargets.add(targetNode);
            trimmedStorages.add(newtargetstorages[i]);
        }
    }
}
if ((closeFile) && !trimmedTargets.isEmpty()) {
    // 将副本信息添加到 DatanodeStorageInfo 中保存
    for (int i = 0; i < trimmedTargets.size(); i++) {
        DatanodeStorageInfo storageInfo =
            trimmedTargets.get(i).getStorageInfo(trimmedStorages.get(i));
        if (storageInfo != null) {
            storageInfo.addBlock(storedBlock);
        }
    }
}

// 在 INode 中记录保存最后一个数据块的数据节点存储的信息
DatanodeStorageInfo[] trimmedStorageInfos =
    blockManager.getDatanodeManager().getDatanodeStorageInfos(
        trimmedTargets.toArray(new DatanodeID[trimmedTargets.size()] ),
        trimmedStorages.toArray(new String[trimmedStorages.size()]));
iFile.setLastBlock(storedBlock, trimmedStorageInfos);
}

if (closeFile) {
    // 调用 closeFileCommitBlocks() 关闭文件
    src = closeFileCommitBlocks(iFile, storedBlock);
} else {
    // 如果不关闭文件, 则在 editlog 中记录
    src = iFile.getFullPathName();
    persistBlocks(src, iFile, false);
}
} finally {
    writeUnlock();
}
}
getEditLog().logSync();
}
```

完成了 `commitBlockSynchronization()` 方法, Namenode 上数据块的信息与 Datanode 上的副本信息也就一致了, 这时整个租约恢复流程也就结束了。

7. 租约恢复——其他方式发起

租约恢复除了可以由 LeaseManager.Monitor 线程发起外，如图 3-53 所示，还有以下三种情况会调用 FSNamesystem.recoverLeaseInternal()方法触发租约恢复操作，这三种情况调用 recoverLeaseInternal()的不同之处在于 force 字段的赋值不同。

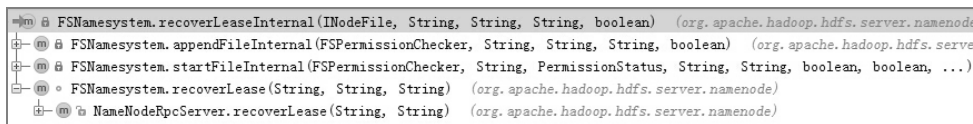


图 3-53 FSNamesystem.recoverLeaseInternal()调用关系

- 客户端通过 ClientProtocol.recoverLease() 远程方法发起租约恢复，最终会由 FSNamesystem.recoverLeaseInternal()响应，这里会将 force 字段置为 true，也就是强制执行 internalReleaseLease()方法关闭文件并释放租约，而不用判断租约是否超过了软限制时间。
- 客户端通过 startFileInternal() 打开一个文件以进行写操作，这时候会调用 recoverLeaseInternal()方法检查是否有别的客户端打开了这个文件，以防止多个客户端同时写这个文件。如果有别的客户端打开了这个文件，recoverLeaseInternal()方法会抛出 AlreadyBeingCreatedException 异常。这里的 force 字段设置为 false，方法会判断原租约持有者是否已经软超时（softLimit），如果超时则进行租约恢复操作释放租约并关闭文件，为文件写操作做准备。
- 客户端通过 appendFileInternal()打开文件进行追加写操作与 startFileInternal()是同一个道理，调用 recoverLeaseInternal()方法检查是否有别的客户端同时打开了这个文件。

```

private void recoverLeaseInternal(INodeFile fileINode,
    String src, String holder, String clientMachine, boolean force)
    throws IOException {
    assert hasWriteLock();
    if (fileINode != null && fileINode.isUnderConstruction()) {
        // 如果文件处于打开的构建状态，那么该文件的租约一定是属于当前的持有者
        Lease lease = leaseManager.getLease(holder);

        // 如果是当前客户端发起的重复创建文件的请求，则抛出异常
        if (!force && lease != null) {
            Lease leaseFile = leaseManager.getLeaseByPath(src);
            if (leaseFile != null && leaseFile.equals(lease)) {
                throw new AlreadyBeingCreatedException(...);
            }
        }
    }

    // 当前租约不存在，或者是客户端发起的租约恢复操作
    // 则通过 HDFS 文件找到源租约
    FileUnderConstructionFeature uc = fileINode.getFileUnderConstructionFeature();
    String clientName = uc.getClientName();
  
```

```
lease = leaseManager.getLease(clientName);
if (lease == null) {
    throw new AlreadyBeingCreatedException();
}
if (force) {
    // 如果是客户端发起的，则强制执行租约恢复
    internalReleaseLease(lease, src, holder);
} else {
    // 如果原租约软超时，则执行租约恢复
    if (lease.expiredSoftLimit()) {
        boolean isClosed = internalReleaseLease(lease, src, null);
        if (!isClosed)
            // 租约恢复正在进行中，则抛出异常
            throw new RecoveryInProgressException(
                "Failed to close file " + src +
                ". Lease recovery is in progress. Try again later.");
    } else {
        final BlockInfo lastBlock = fileINode.getLastBlock();
        if (lastBlock != null
            && lastBlock.getBlockUCState() == BlockUCState.UNDER_RECOVERY) {
            // 租约恢复正在进行中，则抛出异常
            throw new RecoveryInProgressException("Recovery in progress, file ["
                + src + "], " + "lease owner [" + lease.getHolder() + "]");
        } else {
            // 已经有客户端打开了这个文件，并且正在写入，则抛出异常
            throw new AlreadyBeingCreatedException("...");
        }
    }
}
}
```

3.5 缓存管理

Hadoop 2.3.0 版本新增了集中式缓存管理（Centralized Cache Management）功能，允许用户将一些文件和目录保存到 HDFS 缓存中。HDFS 集中式缓存是由分布在 Datanode 上的堆外内存组成的，并且由 Namenode 统一管理。

添加集中式缓存功能的 HDFS 集群具有以下显著的优势。

- 阻止了频繁使用的数据从内存中清除。
- 因为集中式缓存是由 Namenode 统一管理的，所以 HDFS 客户端可以根据数据块的缓存情况调度任务，从而提高了数据块的读性能。
- 数据块被 Datanode 缓存后，客户端就可以使用一个新的更高效的零拷贝机制读取数据块。因为数据块被缓存时已经执行了校验操作，所以使用零拷贝读取数据块的客户端不会有读取开销。

- 可以提高集群的内存利用率。当 Datanode 使用操作系统的 buffer 缓存数据块时，对一个块的重复读会导致该块的 N 个副本全部被送入操作系统的 buffer 中。而使用集中式缓存时，用户可以锁定这 N 个副本中的 M 个，从而节约了 $N-M$ 的内存。

本节首先介绍 HDFS 集中式缓存的概念、架构，然后介绍 Namenode 中负责管理缓存的组件 CacheManager 以及 CacheReplicationMonitor 的实现。

3.5.1 缓存概念

HDFS 集中式缓存有两个主要概念。

- 缓存指令（Cache Directive）：一条缓存指令定义了一个要被缓存的路径（path）。这些路径可以是文件夹或文件。需要注意的是，文件夹的缓存是非递归的，只有在文件夹第一层列出的文件才会被缓存。文件夹也可以指定额外的参数，比如缓存副本因子（replication）、有效期等。缓存副本因子设置了路径的缓存副本数，如果多个缓存指令指向同一个文件，那么就用最大缓存副本因子。
- 缓存池（Cache Pool）：缓存池是一个管理单元，是管理缓存指令的组。缓存池拥有类似 UNIX 的权限，可以限制哪个用户和组可以访问该缓存池。缓存池也可以用于资源管理。它可以设置一个最大限制值，限制写入缓存池中指令的字节数。

3.5.2 缓存管理命令

HDFS 为管理员和用户提供了“hdfs cacheadmin”命令管理集中式缓存，包括缓存指令控制和缓存池控制两个部分。

- 缓存指令控制：管理员可以调用“hdfs cacheadmin -addDirective”命令缓存指定路径；调用“hdfs cacheadmin -removeDirective”命令删除指定 id 对应的缓存；调用“hdfs cacheadmin -removeDirectives”命令删除指定路径的缓存；调用“hdfs cacheadmin -listDirectives”命令显示当前所有的缓存。由于篇幅原因，这里就不列出所有命令的参数了。
- 缓存池控制：管理员可以调用“hdfs cacheadmin -addPool”命令创建一个缓存池；调用“hdfs cacheadmin -modifyPool”命令修改一个缓存池的配置；调用“hdfs cacheadmin -removePool”命令删除一个缓存池。

3.5.3 HDFS 集中式缓存架构

如图 3-54 所示，用户可以通过“hdfs cacheadmin”命令或者 HDFS API 向 Namenode 发送缓存指令（Cache Directive），Namenode 的 CacheManager 类会将缓存指令保存到内存的指定数据结构（CacheManager 的 directivesById、directivesByPath 字段）中，同时在 fsimage 和 editlog 文件中记录该缓存指令。之后 Namenode 的 CacheReplicationMonitor 类会周期性扫描命名空间和活跃的缓存指令，以确定需要缓存或删除缓存的数据块，并向 Datanode 分配缓存

任务。

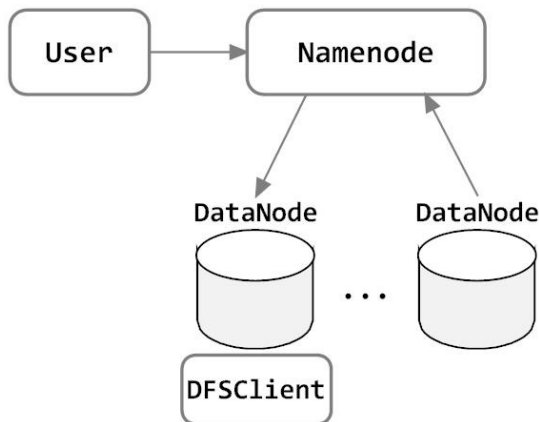


图 3-54 HDFS 集中式缓存架构示意图

Namenode 还负责管理集群中所有 Datanode 的堆外缓存，Datanode 会周期性向 Namenode 发送缓存报告，而 Namenode 会通过心跳响应向 Datanode 下发缓存指令（添加缓存或者删除缓存）。

DFSCClient 读取数据块时会向 Namenode 发送 `ClientProtocol.getBlockLocations` 请求获取数据块的位置信息，Namenode 除了返回数据块的位置信息外，还会返回该数据块的缓存信息，这样 DFSCClient 就可以执行本地零拷贝读取缓存数据块了，从而提高了读取效率。

3.5.4 CacheManager 类实现

CacheManager 类是 Namenode 管理集中式缓存的核心组件，它管理着分布在 HDFS 集群中 Datanode 上的所有缓存数据块，同时负责响应“`hdfs cacheadmin`”命令或者 HDFS API 发送的缓存管理命令（请参考缓存管理命令小节）。

如下代码所示，CacheManager 定义了以下字段。

- `directivesById`: 以缓存指令 id 为 key，保存所有的缓存指令。
- `directivesByPath`: 以路径为 key，保存所有的缓存指令。
- `cachePools`: 以缓存池名称为 key，保存所有的缓存池。
- `monitor`: `CacheReplicationMonitor` 对象，负责扫描命名空间和活跃的缓存指令，以确定需要缓存或删除缓存的数据块，并向 Datanode 分配缓存任务。

```
private final TreeMap<Long, CacheDirective> directivesById =  
    new TreeMap<Long, CacheDirective>();  
  
private final TreeMap<String, List<CacheDirective>> directivesByPath =  
    new TreeMap<String, List<CacheDirective>>();
```

```
private final TreeMap<String, CachePool> cachePools =
    new TreeMap<String, CachePool>();

private CacheReplicationMonitor monitor;
```

客户端会调用 `ClientProtocol.addCachePool()` 方法创建一个缓存池，`NameNodeRpcServer` 接收了这个请求后，会将创建缓存池请求的参数封装到一个 `CachePoolInfo` 对象中（包括 `owner`、`group`、`mode`、`limit`、`maxTtl` 等参数），然后调用 `CacheManager.addCachePool()` 方法响应这个请求。`CacheManager.addCachePool()` 方法首先会验证请求参数（`info` 变量保存）是否合法，然后检查 `CacheManager.cachePools` 集合中是否已经保存了这个缓存池。如果 `CacheManager` 中没有这个缓存池的信息，则调用 `CachePool.createFromInfoAndDefaults()` 方法根据请求参数创建一个新的 `CachePool` 对象。`createFromInfoAndDefaults()` 方法会将请求中没有设置的参数用默认值补齐，然后构造 `CachePool` 对象。成功创建 `CachePool` 对象后，`addCachePool()` 方法会将 `CachePool` 对象放入 `cachePools` 字段中保存。至此，添加缓存池的操作完成。`CacheManager.addCachePool()` 方法的代码如下：

```
public CachePoolInfo addCachePool(CachePoolInfo info)
    throws IOException {
    assert namesystem.hasWriteLock();
    CachePool pool;
    try {
        CachePoolInfo.validate(info); // 验证请求的参数是否合法
        String poolName = info.getPoolName();

        // 确认 cachePools 字段中不包含这个缓存池的信息
        pool = cachePools.get(poolName);
        if (pool != null) { // 如果 cachePools 字段中已经有了信息，则抛出异常
            throw new InvalidRequestException("Cache pool " + poolName
                + " already exists.");
        }

        // 构造 CachePool 对象，对于没有设置的参数，使用默认值补齐
        pool = CachePool.createFromInfoAndDefaults(info);
        // 将新构造的 CachePool 对象放入 cachePools 集合中
        cachePools.put(pool.getPoolName(), pool);
    } catch (IOException e) {
        LOG.info("addCachePool of " + info + " failed: ", e);
        throw e;
    }
    LOG.info("addCachePool of {} successful.", info);
    return pool.getInfo(true);
}
```

成功地创建了缓存池之后，客户端就会调用 `ClientProtocol.addCacheDirective()` 方法将一个路径添加到缓存中。`NameNodeRpcServer` 接收了这个请求后，会将创建缓存请求的参数封装到一个 `CacheDirectiveInfo` 对象中，然后调用 `CacheManager.addInternal()` 方法响应这个请求。

如下代码所示，`CacheManager.addInternal()` 会将缓存指令对象加入 `directivesById` 以及 `directivesByPath` 集合中保存，然后更新缓存池的统计信息。之后 `addInternal()` 方法会调用 `setNeedsRescan()` 方法触发 `CacheReplicationMonitor` 执行 `rescan()` 操作。

```
private void addInternal(CacheDirective directive, CachePool pool) {
    boolean addedDirective = pool.getDirectiveList().add(directive);
    assert addedDirective;
    // 将缓存指令对象加入 directivesById 集合中保存
    directivesById.put(directive.getId(), directive);
    String path = directive.getPath();
    // 将缓存指令对象加入 directivesByPath 集合中保存
    List<CacheDirective> directives = directivesByPath.get(path);
    if (directives == null) {
        directives = new ArrayList<CacheDirective>(1);
        directivesByPath.put(path, directives);
    }
    directives.add(directive);
    // 更新缓存池的统计信息
    CacheDirectiveStats stats =
        computeNeeded(directive.getPath(), directive.getReplication());
    directive.addBytesNeeded(stats.getBytesNeeded());
    directive.addFilesNeeded(directive.getFilesNeeded());

    // 触发 CacheReplicationMonitor 执行 rescan() 操作
    setNeedsRescan();
}
```

可以看到，`addInternal()` 方法只是更新了 `CacheManager` 的 `directivesById`、`directivesByPath` 以及 `cachePools` 等字段维护的缓存信息，之后并没有对 `Datanode` 下发任何缓存指令（添加、删除缓存），而是调用 `setNeedsRescan()` 方法触发 `CacheReplicationMonitor` 执行缓存操作。我们在下一节介绍 `CacheReplicationMonitor` 的实现。

3.5.5 CacheReplicationMonitor

`CacheReplicationMonitor` 是一个线程类，它会在 `Namenode` 启动时以及以固定的间隔扫描命名空间和活跃的缓存指令，以确定需要缓存或删除缓存的数据块，之后向 `Datanode` 下发缓存指令（这里的间隔是由 `dfs.namenode.path.based.cache.refresh.interval.ms` 配置项配置的，默认为 30 秒）。

`CacheReplicationMonitor` 会循环调用 `rescan()` 方法执行扫描逻辑，`rescan()` 方法的代码如下所示，它会调用 `rescanCacheDirectives()` 方法遍历所有的缓存指令（存储在 `CacheManager.directivesByPath` 字段中），并将缓存指令路径中包含的数据块加入到 `CacheReplicationMonitor.cachedBlocks` 集合中等待进一步操作。之后 `rescan()` 方法会调用 `rescanCachedBlockMap()` 遍历 `CacheReplicationMonitor.cachedBlocks` 集合，判断这些数据块是执行 `cache` 操作还是 `uncache` 操作。对于需要执行 `cache` 操作的数据块，`rescanCachedBlockMap()` 会调用

CacheReplicationMonitor.addNewPendingCached()方法为每个等待 cache 的数据块选择一个合适的 Datanode (从保存了该数据块副本的所有 Datanode 中挑选一个可用内存最多的), 之后将该数据块加入该 Datanode 对应的 DatanodeDescriptor 对象的 pendingCached 列表中。而对于需要执行 uncache 操作的数据块, rescanCachedBlockMap()会调用 CacheReplicationMonitor.addNewPendingUncached()方法从缓存了该数据块的 Datanode 中随机选出一个节点执行 uncache 操作, 也就是将数据块加入该 Datanode 对应的 DatanodeDescriptor 对象的 pendingUncached 列表中。

```
private void rescan() throws InterruptedException {
    scannedDirectives = 0;
    scannedBlocks = 0;
    try {
        namesystem.writeLock();
        // ...
        rescanCacheDirectives();
        rescanCachedBlockMap();
        blockManager.getDatanodeManager().resetLastCachingDirectiveSentTime();
    } finally {
        namesystem.writeUnlock();
    }
}
```

将数据块加入 DatanodeDescriptor 的 pendingCached 和 pendingUncached 列表中后, Namenode 就会在心跳处理流程中生成名字节点指令(请参考数据节点管理的名字节点指令生成小节), 并通过心跳响应发送给 Datanode。Datanode 接受指令之后, 会执行缓存以及删除缓存操作(请参考第4章的 FSDatasetImpl 小节)。至此, Namenode 端处理缓存数据块的逻辑就结束了。DFSClient 通过零拷贝模式读取缓存数据块的实现, 请读者参考第5章的零拷贝读取相关小节。

3.6 ClientProtocol 实现

Namenode 作为 HDFS 的大脑实现了 ClientProtocol、DatanodeProtocol 以及 NamenodeProtocol 等远程接口, 在前面的介绍中我们已经分析了 DatanodeProtocol 以及 NamenodeProtocol 的大部分实现, 本节则重点介绍 ClientProtocol 中与读写相关的方法在 Namenode 中的实现, 包括创建文件、追加写文件、创建新的数据块、放弃数据块以及关闭文件等操作。

3.6.1 创建文件

如第1章的 HDFS 客户端写流程小节中所介绍的, DFSClient 写数据的第一步就是创建一个新的 HDFS 文件执行写操作, 或者是打开一个已有的 HDFS 文件执行追加写操作。本节先介绍创建文件部分。

创建 HDFS 文件的调用流程如图 3-55 所示，客户端通过调用 `ClientProtocol.create()` 方法创建一个新的文件，这个调用由 `NamenodeRpcServer.create()` 方法响应，`create()` 方法会调用 `FSNamesystem.startFileInt()`，`startFileInt()` 方法会级联调用 `FSNamesystem.startFileInternal()` 来创建一个新的文件。

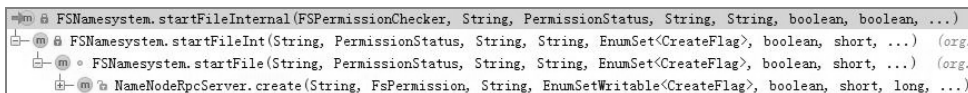


图 3-55 `ClientProtocol.create()` 调用流程图

下面我们看一下 `FSNamesystem.startFileInternal()` 方法的实现。首先看一下方法参数中比较重要的几个标志位：`overwrite` 用于指示如果目标文件 `src` 已经存在了，是否覆盖这个文件；`create` 用于指示 `src` 文件不存在时，是否创建这个文件；`createParent` 用于指示目标文件的父目录不存在时，是否创建目录。

`startFileInternal()` 方法会在文件系统目录树的 `src` 路径上创建一个 `INodeFileUnderConstruction` 对象，并在租约管理器中添加租约。当然，这个方法还包括很多判断条件，整个方法的流程如下所示。

- 判断 `src` 路径在目录树中是否指向一个已经存在的目录，如果是则抛出异常。
- 检查权限，在目标文件的父目录上必须有写权限，如果 `overwrite` 参数为 `true`，则在目标文件上有写权限。
- `createParent` 标识位为 `false` 时，检查目标文件的父目录以及祖先目录是否存在。
- 如果目标文件不存在且 `create` 标识位为 `false`，或者目标文件存在且 `overwrite` 参数为 `false` 时，则抛出异常。
- 如果是覆盖写文件（`overwrite` 参数为 `true`），则调用 `FSDirectory.delete()` 从文件系统目录树中删除这个 HDFS 文件，之后调用 `removePathAndBlocks()` 方法删除租约，从 `INodeMap` 中删除 `INode`，从 `Namenode` 和 `Datanode` 上删除数据块。
- 在目标文件路径上创建一个 `INodeFileUnderConstruction` 对象并插入目录树，之后在租约管理器中添加租约。

`startFileInternal()` 方法的代码如下：

```
private BlocksMapUpdateInfo startFileInternal(FSPermissionChecker pc,
    String src, PermissionStatus permissions, String holder,
    String clientMachine, boolean create, boolean overwrite,
    boolean createParent, short replication, long blockSize,
    boolean isLazyPersist, CipherSuite suite, CryptoProtocolVersion version,
    EncryptedKeyVersion edek, boolean logRetryEntry)
    throws FileAlreadyExistsException, AccessControlException,
    UnresolvedLinkException, FileNotFoundException,
    ParentNotDirectoryException, RetryStartFileException, IOException {
    assert hasWriteLock();
    // 如果目标文件 src 在目录树中已经存在，并且是一个目录，则抛出异常
    final INodesInPath iip = dir.getInodesInPath4Write(src);
```

```

final INode inode = iip.getLastINode();
if (inode != null && inode.isDirectory()) {
    throw new FileAlreadyExistsException(src +
        " already exists as a directory");
}

// FileEncryptionInfo 部分省略

// 检查权限, 在父目录上必须有写权限, 如果是覆盖操作, 则在目标文件上有写权限
final INodeFile myFile = INodeFile.valueOf(inode, src, true);
if (isPermissionEnabled) {
    if (overwrite && myFile != null) {
        checkPathAccess(pc, src, FsAction.WRITE);
    }
    checkAncestorAccess(pc, src, FsAction.WRITE);
}

// createParent 为 false 时, 检查目标文件的父目录以及祖先目录是否存在
if (!createParent) {
    verifyParentDir(src);
}

try {
    BlocksMapUpdateInfo toRemoveBlocks = null;
    // 如果该文件不存在, 且用户设置了不创建这个文件, 则抛出异常
    if (myFile == null) {
        if (!create) {
            throw new FileNotFoundException("Can't overwrite non-existent " +
                src + " for client " + clientMachine);
        }
    } else {
        // 如果文件存在, 则首先删除这个文件、文件对应的数据块以及租约
        if (overwrite) {
            toRemoveBlocks = new BlocksMapUpdateInfo();
            List<INode> toRemoveINodes = new ChunkedArrayList<INode>();
            long ret = dir.delete(src, toRemoveBlocks, toRemoveINodes, now());
            if (ret >= 0) {
                incrDeletedFileCount(ret);
                removePathAndBlocks(src, null, toRemoveINodes, true);
            }
        } else {
            // 如果文件存在, 且用户设置了不覆盖这个文件, 则抛出异常
            recoverLeaseInternal(myFile, src, holder, clientMachine, false);
            throw new FileAlreadyExistsException(src + " for client " +
                clientMachine + " already exists");
        }
    }
}
// ...

```

```
// 创建文件
Path parent = new Path(src).getParent();
if (parent != null && mkdirsRecursively(parent.toString(),
    permissions, true, now())) {
    newNode = dir.addFile(src, permissions, replication, blockSize,
        holder, clientMachine);
}
// 添加租约
leaseManager.addLease(newNode.getFileUnderConstructionFeature()
    .getClientName(), src);

// 在 editlog 中记录这个操作
getEditLog().logOpenFile(src, newNode, overwrite, logRetryEntry);
return toRemoveBlocks;
} catch (IOException ie) {
    throw ie;
}
}
```

3.6.2 追加写文件

当客户端需要重新打开一个文件进行追加写操作时，会调用 `ClientProtocol.append()` 方法重新打开一个 HDFS 文件。需要注意的是，`append()` 方法返回的是这个 HDFS 文件的最后一个未写满数据块的位置信息 (`LocateBlock`)，有了这个信息后，客户端就可以建立数据流管道对这个数据块进行追加写操作了。通过图 3-56 所示的调用顺序我们知道，追加写文件的操作是在 `FSNamesystem.appendFileInternal()` 方法中实现的。

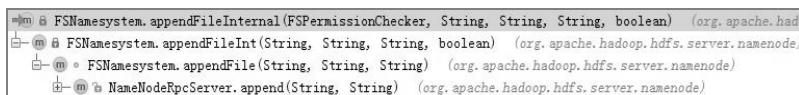


图 3-56 `ClientProtocol.append()` 调用流程图

`appendFileInternal()` 方法会返回这个文件的最后一个未写满的数据块，客户端会通过该数据块的位置信息建立输出流管道，并将数据追加写入这个最后的未写满的数据块中。如果当前文件的最后一个数据块正好写满，则方法返回空，客户端会调用 `ClientProtocol.addBlock()` 方法从 `Namenode` 申请一个新的数据块写入。

`appendFileInternal()` 方法的实现比较简单，它首先做了一系列检查，检查目标文件在 HDFS 中是否是一个目录、检查权限、检查目标文件是否存在、检查文件最后一个数据块的副本数是否足够，然后调用 `recoverLeaseInternal()` 检查租约（请参考租约管理的租约恢复小节），最后调用 `prepareFileForWrite()` 返回文件最后一个未写满数据块的位置信息，如果最后一个数据块写满了，则返回 `null`。

```
private LocatedBlock appendFileInternal(FSPermissionChecker pc, String src,
    String holder, String clientMachine, boolean logRetryCache)
```

```

        throws AccessControlException, UnresolvedLinkException,
        FileNotFoundException, IOException {
    assert hasWriteLock();
    // 检查目标文件在 HDFS 中是不是一个目录
    final INodesInPath iip = dir.getINodesInPath4Write(src);
    final INode inode = iip.getLastINode();
    if (inode != null && inode.isDirectory()) {
        throw new FileAlreadyExistsException("Cannot append to directory " + src
            + "; already exists as a directory.");
    }
    // 检查权限
    if (isPermissionEnabled) {
        checkPathAccess(pc, src, FsAction.WRITE);
    }

    // 检查目标文件是否存在
    try {
        if (inode == null) {
            throw new FileNotFoundException("failed to append to non-existent file "
                + src + " for client " + clientMachine);
        }

        // 在打开一个已有的文件前, 首先调用 recoverLeaseInternal() 检查租约
        recoverLeaseInternal(myFile, src, holder, clientMachine, false);

        // recoverLeaseInternal() 方法可能会通过 finalizeINodeFileUnderConstruction
        // 更新命名空间中的 INode, 所以这里需要刷新一下 INode 的引用
        myFile = INodeFile.valueOf(dir.getINode(src), src, true);
        final BlockInfo lastBlock = myFile.getLastBlock();

        // 判断最后一个数据块是否有足够的副本数
        if (lastBlock != null && lastBlock.isComplete() &&
            !getBlockManager().isSufficientlyReplicated(lastBlock)) {
            throw new IOException("append: lastBlock=" + lastBlock +
                " of src=" + src + " is not sufficiently replicated yet.");
        }

        // 调用 prepareFileForWrite() 获取文件最后一个未写满的数据块
        return prepareFileForWrite(src, myFile, holder, clientMachine, true,
            iip.getLatestSnapshotId(), logRetryCache);
    } catch (IOException ie) {
        throw ie;
    }
}

```

`appendFileInternal()` 会调用 `prepareFileForWrite()` 方法获取文件中最后一个未写满的数据块的位置信息。`prepareFileForWrite()` 方法的逻辑也比较简单, 它首先把文件状态转换为构建中状态, 也就是在 `INode` 上添加 `UnderConstructionFeature` 的特性, 然后在租约管理器中添加租

约信息,最后调用 `BlockManager.convertLastBlockToUnderConstruction()`方法返回文件中最后一个未写满的数据块。

```
LocatedBlock prepareFileForWrite(String src, INodeFile file,
                                String leaseHolder, String clientMachine,
                                boolean writeToEditLog,
                                int latestSnapshot, boolean logRetryCache)
    throws IOException {
    file.recordModification(latestSnapshot);

    // 在 INodeFile 上添加 UnderConstructionFeature 的特性
    final INodeFile cons = file.toUnderConstruction(leaseHolder, clientMachine);
    // 在租约管理器中添加租约信息
    leaseManager.addLease(cons.getFileUnderConstructionFeature()
        .getClientName(), src);

    // 调用 convertLastBlockToUnderConstruction() 返回最后一个未写满的数据块
    LocatedBlock ret = blockManager.convertLastBlockToUnderConstruction(cons);
    // ...

    // 在 editlog 中记录这个操作
    if (writeToEditLog) {
        getEditLog().logOpenFile(src, cons, false, logRetryCache);
    }
    return ret;
}
```

`BlockManager.convertLastBlockToUnderConstruction()`方法除了返回文件中最后一个未写满的数据块外,还会改变最后一个数据块的状态,将数据块改为构建中状态。由于要执行追加写操作,数据块的状态会一直发生改变,所以这个方法还会停止所有对这个数据块的复制请求、删除请求,也就是从 `BlockManager` 的 `pendingReplications`、`neededReplications` 以及 `invalidateBlocks` 队列中删除这个数据块的信息。`convertLastBlockToUnderConstruction()`方法的代码如下:

```
public LocatedBlock convertLastBlockToUnderConstruction(
    BlockCollection bc) throws IOException {
    BlockInfo oldBlock = bc.getLastBlock();
    // 如果最后一个数据块写满了,或者这个文件是一个刚刚创建的空文件,则返回 null
    if(oldBlock == null ||
        bc.getPreferredBlockSize() == oldBlock.getNumBytes())
        return null;

    // 获取存储这个数据块的 Datanode 信息
    DatanodeStorageInfo[] targets = getStorages(oldBlock);

    // 将最后一个数据块的状态更新为构建中状态
    BlockInfoUnderConstruction ucBlock = bc.setLastBlock(oldBlock, targets);
    blocksMap.replaceBlock(ucBlock);
}
```

```

// 从复制队列中删除这个数据块相关的所有请求
NumberReplicas replicas = countNodes(ucBlock);
neededReplications.remove(ucBlock, replicas.liveReplicas(),
    replicas.decommissionedReplicas(), getReplication(ucBlock));
pendingReplications.remove(ucBlock);

// 从删除队列中删除这个数据块相关的所有请求
for (DatanodeStorageInfo storage : targets) {
    invalidateBlocks.remove(storage.getDatanodeDescriptor(), oldBlock);
}

// 构造 LocatedBlock 并返回
final long fileLength = bc.computeContentSummary().getLength();
final long pos = fileLength - ucBlock.getNumBytes();
return createLocatedBlock(ucBlock, pos, AccessMode.WRITE);
}

```

当客户端接收了 Namenode 返回的最后一个数据块的位置信息后，就会建立数据流管道向 Datanode 追加写数据。当数据节点成功地接收了一个完整的数据块之后，就会通过 DatanodeProtocol.blockReceivedAndDeleted()方法向 Namenode 汇报，这部分内容我们在数据块管理小节中已经学习过了。对于客户端，完成一个数据块的写操作后，会向 Namenode 申请新的数据块，我们在下面的小节中介绍创建新的数据块操作。

3.6.3 创建新的数据块

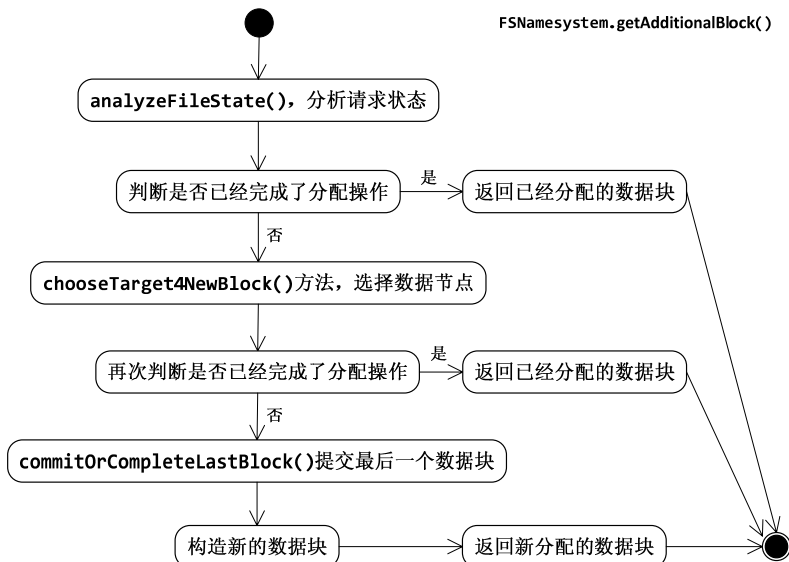
DFSClient 在写数据时，首先会调用 ClientProtocol.create()方法在 Namenode 的文件系统目录树中创建一个 HDFS 文件，这个请求在 Namenode 侧会由 FSNamesystem.startFile()方法响应。startFile()方法会创建 INode 对象并将 INode 对象加入 FSDirectory 中，然后添加租约，并将操作写入 editlog 中。

成功创建 INode 对象后，DFSClient 会调用 ClientProtocol.addBlock()请求分配新的数据块，Namenode 会调用 FSNamesystem.getAdditionalBlock()方法响应分配请求并返回新申请的数据块，以及存储这个数据块副本的 Datanode 信息。本节就介绍 getAdditionalBlock()方法的实现。

图 3-57 给出了 getAdditionalBlock()方法的逻辑流程图。

- 首先调用 analyzeFileState()方法检查 Namenode 是否有写权限，是否处于安全模式中，以及当前文件系统中的对象是否过多。然后对当前 Namenode 中记录的该文件的最后一个数据块与 Client 上报的最后一个数据块进行比较，最后判断是否发生了 RPC 请求的重传，或者请求异常等情况。
- 对于 Namenode 已经完成了数据块分配操作，但是 Client 未能正确收到响应而重发请求的情况，直接返回 Namenode 中保存的上一次成功分配的数据块。
- 在其他情况下，首先调用 chooseTarget4NewBlock()为新分配的数据块选择保存副本的 Datanode。

- 再次调用 `analyzeFileState()` 方法以防止 `chooseTarget4NewBlock()` 方法执行期间 Namenode 状态发生改变, 例如在调用 `chooseTarget4NewBlock()` 方法时, 上一次触发的分配请求执行完成, 这时无须再次分配新的数据块, 返回上一次分配的数据块即可。

图 3-57 `getAdditionalBlock()` 方法的逻辑流程图

- 由于 Client 调用 `ClientProtocol.addBlock()` 请求时, 还会顺带提交文件的最后一个数据块, 所以 `getAdditionalBlock()` 会调用 `commitOrCompleteLastBlock()` 方法提交文件的最后一个数据块, 如果这个数据块满足最小的副本系数, 则将这个数据块的状态转换为 `COMPLETE`。
- 最终情况是, 创建新的数据块并保存在 `Inode` 对象中, 之后将新创建的数据块返回至 Client。

```

LocatedBlock getAdditionalBlock(String src, long fileId, String clientName,
    ExtendedBlock previous, Set<Node> excludedNodes,
    List<String> favoredNodes)
    throws LeaseExpiredException, NotReplicatedYetException,
    QuotaExceededException, SafeModeException, UnresolvedLinkException,
    IOException {

```

```

    // ...
    readLock();
    try {
        // 调用 analyzeFileState() 方法分析当前请求状态
        // 如果是重试请求, 并且可以将上一次分配的数据块返回, 则保存在 onRetryBlock 中
        LocatedBlock[] onRetryBlock = new LocatedBlock[1];
        FileState fileState = analyzeFileState(
            src, fileId, clientName, previous, onRetryBlock);
        final InodeFile pendingFile = fileState.inode;

```



```

src = fileState.path;

if (onRetryBlock[0] != null && onRetryBlock[0].getLocations().length > 0) {
    // 这是一次重试请求, 将 onRetryBlock 保存的上一次分配的数据块返回
    return onRetryBlock[0];
}
// ...
} finally {
    readUnlock();
}

// 调用 chooseTarget4NewBlock() 选择保存数据块副本的数据节点
final DatanodeStorageInfo targets[] = getBlockManager().chooseTarget4NewBlock(
    src, replication, clientNode, excludedNodes, blockSize, favoredNodes,
    storagePolicyID);

// 第二部分
// 分配一个新的数据块, 并将这个数据块添加到 INode 以及 BlocksMap 中
try {
    checkOperation(OperationCategory.WRITE);
    // 再次调用 analyzeFileState()
    // 因为在执行 chooseTarget4NewBlock() 时 Namenode 状态可能发生改变
    LocatedBlock[] onRetryBlock = new LocatedBlock[1];
    FileState fileState =
        analyzeFileState(src, fileId, clientName, previous, onRetryBlock);
    final INodeFile pendingFile = fileState.inode;
    src = fileState.path;

    // 之前触发的分配请求完成, 新的数据块保存在 onRetryBlock 变量中, 直接返回
    if (onRetryBlock[0] != null) {
        if (onRetryBlock[0].getLocations().length > 0) {
            // 上一次请求成功构造了新的数据块, 直接返回.
            return onRetryBlock[0];
        } else {
            BlockInfo lastBlockInFile = pendingFile.getLastBlock();
            ((BlockInfoUnderConstruction) lastBlockInFile)
                .setExpectedLocations(targets);
            offset = pendingFile.computeFileSize();
            // 设置保存数据块副本系数的数据节点, 然后返回数据块
            return makeLocatedBlock(lastBlockInFile, targets, offset);
        }
    }
}

// 提交最后一个数据块
// 如果该数据块的副本系数达到了最小要求, 则设置该数据块状态为 COMPLETE
commitOrCompleteLastBlock(pendingFile,
    ExtendedBlock.getLocalBlock(previous));
// 产生新的数据块, 并保存至 INode 中
newBlock = createNewBlock();
INodesInPath inodesInPath = INodesInPath.fromINode(pendingFile);

```

```
        saveAllocatedBlock(src, inodesInPath, newBlock, targets);

        persistNewBlock(src, pendingFile);
        offset = pendingFile.computeFileSize();
    } finally {
        writeUnlock();
    }
    getEditLog().logSync();

    // 返回新创建的数据块
    return makeLocatedBlock(newBlock, targets, offset);
}
```

从上面的代码分析我们可以看到，`getAdditionalBlock()`分别调用了 `analyzeFileState()`、`chooseTarget4NewBlock()`、`commitOrCompleteLastBlock()`等方法配合执行数据块的分配操作，下面我们就一一学习这些方法的实现。

1. 分析状态——`analyzeFileState()`

`getAdditionalBlock()`方法会首先调用 `analyzeFileState()`方法分析当前数据块分配请求的类型，并且将已经分配好的数据块保存在 `onRetryBlock` 参数中。

图 3-58 给出了 `analyzeFileState()`方法的执行逻辑。`analyzeFileState()`方法首先进行一系列判断操作，判断是否有写操作权限，判断 Namenode 是否处于安全模式中，检查文件系统中保存的对象是否太多，检查文件的租约。然后 `analyzeFileState()`会将 Client 通过 `ClientProtocol.addBlock()`方法汇报的最后一个数据块 `previousBlock` 与 Namenode 内存中记录的文件最后一个数据块 `lastBlockInFile` 进行比较。

- 如果 `previousBlock==null`，也就是 `addBlock()`方法并未携带文件最后一个数据块的信息。这种情况可能是 Client 调用 `ClientProtocol.append()`方法申请追加写文件，而文件的最后一个数据块正好写满，Client 就会调用 `addBlock()`方法申请新的数据块。这时 `analyzeFileState()`方法无须执行任何操作，`getAdditionalBlock()`方法正常执行数据块分配操作即可。
- 如果 `previousBlock` 信息与 `penultimateBlock` 信息匹配，`penultimateBlock` 是 Namenode 记录的文件倒数第二个数据块的信息。这种情况是 Namenode 已经成功地为 Client 分配了数据块，但是响应信息并未送回 Client，所以 Client 重发了请求。对于这种情况，由于 Namenode 已经成功地为 Client 分配了数据块，并且 Client 没有向新分配的数据块写入任何数据，所以 `analyzeFileState()`方法会将分配的数据块保存至 `onRetryBlock` 参数中，`getAdditionalBlock()`方法可以直接将 `onRetryBlock` 中保存的数据块再次返回给 Client，而无须构造新的数据块。
- `previousBlock` 信息与 `lastBlockInFile` 信息不匹配，这是异常的情况，不应该出现，`getAdditionalBlock()`方法会直接抛出异常。
- Namenode 正在执行分配数据块操作，例如正在调用 `chooseTarget()`方法，这时 Client 由于请求超时而重新发送了请求。对于重发的请求，`analyzeFileState()`无须进行任何

操作，因为之前的请求并未改变命名空间的状态。

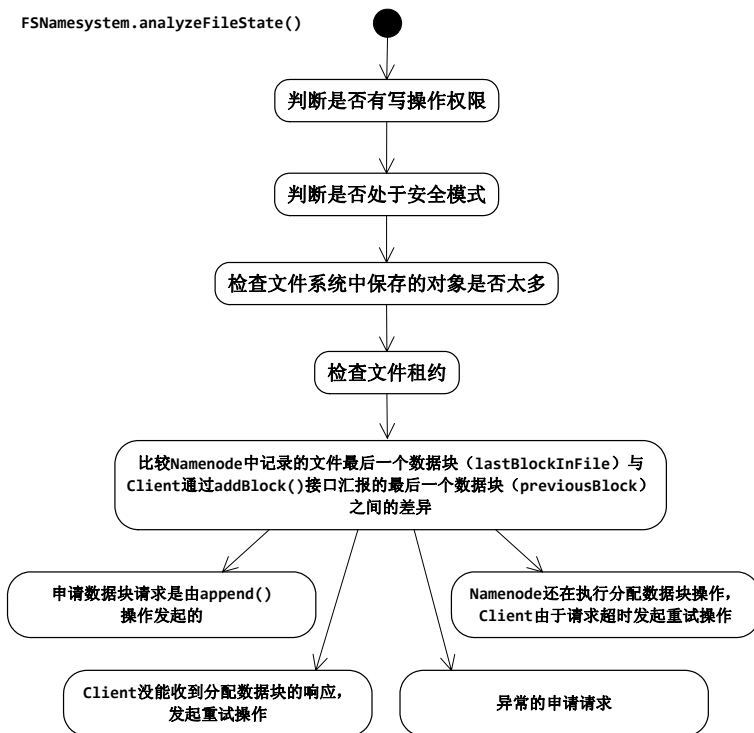


图 3-58 `analyzeFileState()`方法的执行逻辑图

2. 分配数据节点——`chooseTarget4NewBlock()`

`FSNamesystem.getAdditionalBlock()` 方法在分配数据块时，会调用 `BlockManager.chooseTarget4NewBlock() -> BlockPlacementPolicyDefault.chooseTarget()` 方法为指定数据块选择保存这个数据块副本的 `Datanode`。这一节我们就重点了解 `chooseTarget()` 的逻辑实现。

`chooseTarget()` 的逻辑实现可以参考图 3-59。

- 如果 Client 不是一个 `DataNode`，则在集群范围内随机选择一个节点作为第一个节点。
- 如果 Client 是一个 `DataNode`，则判断 Client 所在的本地数据节点是否符合存储数据块的要求，如果符合，则第一个节点分配完毕；如果该数据节点不符目标节点要求，则在 Client 同一个机架范围内寻找，如果找到目标节点，则第一个节点分配完毕；如果在同一个机架内未找到符合要求的目标节点，则在集群内随机分配一个节点，找到则第一个节点分配完毕；否则分配失败。
- 如果已经成功分配第一个数据节点，则在与第一个分配节点不同机架的远程机架内寻找第二个目标节点。如果符合要求，则第二个节点分配完毕；如果在远程机架内未找到符合要求的目标节点，则在第一个分配节点的机架内寻找，如果找到则第二

个节点分配完毕；否则第二个节点分配失败。

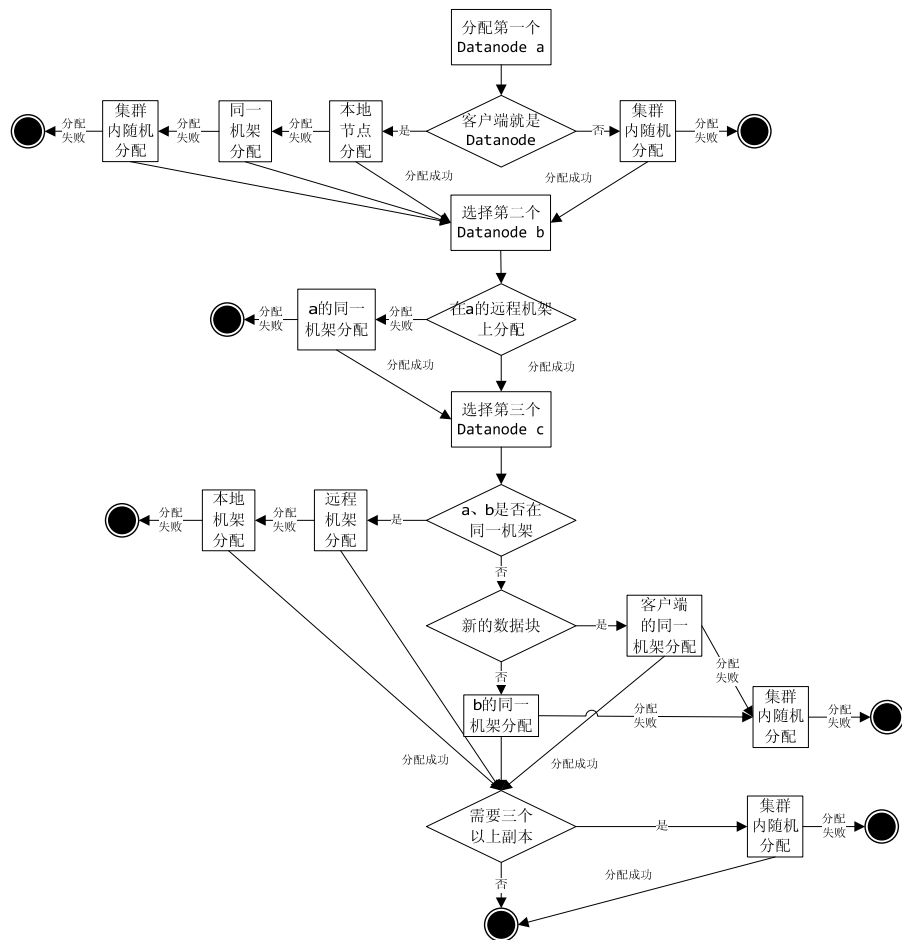


图 3-59 chooseTarget4NewBlock()方法逻辑图

- 如果前两个节点分配成功，则准备分配第三个副本的目标节点。首先判断前两个节点是否在同一个机架内，如果是，则在远程机架内寻找目标节点，找到则第三个节点分配完毕；如果前两个节点在不同的机架内，且当前数据块为新分配的数据块，则在与 Client 相同的机架内寻找。如果当前数据块为已有的数据块，则在第二个节点的机架内分配。如果找到则第三个节点分配完毕，未找到则在集群中随机分配一个节点；否则第三个节点分配失败。
- 如果需要分配的节点数目大于三个，则在集群范围内随机寻找节点。

在 chooseTarget4NewBlock()逻辑中，如何判断一个数据节点满足保存数据块副本的要求呢？这里是通过 BlockPlacementPolicyDefault.isGoodTarget()方法判断的，如果这个方法返回

true, 则当前数据节点可以作为数据块副本的保存节点。isGoodTarget()方法的判断逻辑如图 3-60 所示。

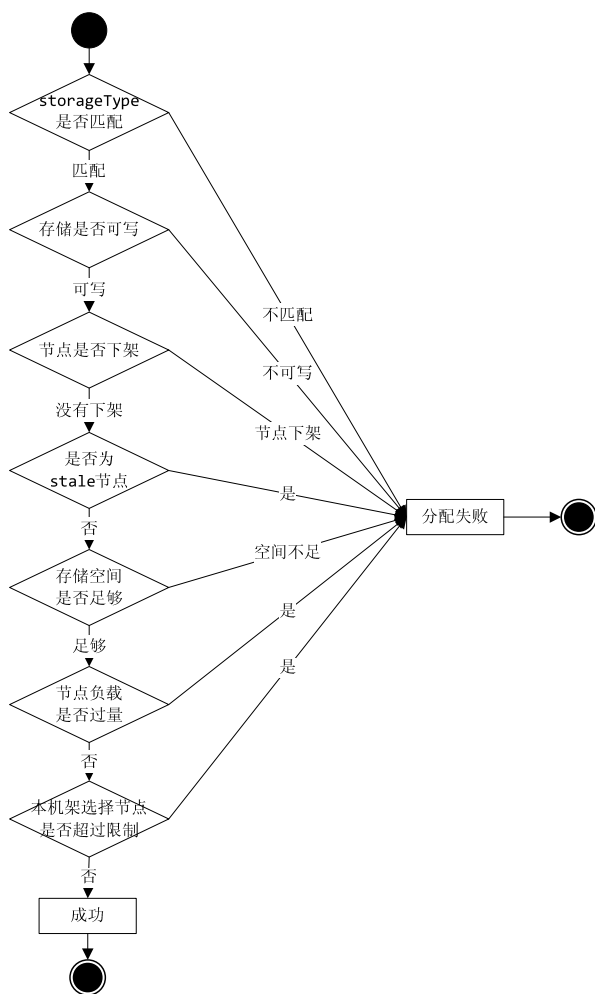


图 3-60 isGoodTarget()方法的判断逻辑图

在 HDFS 2.6 版本以后, 对于数据块管理的粒度从 Datanode 细化到了 Storage, 那么对于数据块副本存储的判断也应该细化到该 Datanode 上的指定 Storage, 所以 isGoodTarget()的判断分为两个部分。

- 存储的判断: 判断该 Datanode 上指定存储的类型是匹配 (例如 SSD、普通的 DISK 等), 该存储是否可写, 该存储是否是有效的存储。
- 存储所在的 Datanode 的判断: 判断数据节点是否为退役的节点, 该节点是否为 stale 状态 (超过 stale 时长未发送心跳), 该节点的存储空间是否足够, 该节点的网络负

载是否过量，该节点所在的机架节点是否负载过大。

满足了上述所有条件之后，该 Datanode 的 Storage 就可以作为选择结果正确返回了。

3. 提交上一个数据块——commitOrCompleteLastBlock()

如图 3-61 所示，commitOrCompleteLastBlock()方法会在以下三种情况下调用。

- 客户端完成对一个文件的写入操作之后，就会调用 commitOrCompleteLastBlock()方法提交该文件的最后一个数据块。
- 客户端申请一个新的数据块时，会调用 commitOrCompleteLastBlock()方法提交上一个数据块。
- Datanode 执行租约恢复操作时，会在 DatanodeProtocol.commitBlockSynchronization()方法中调用 commitOrCompleteLastBlock()关闭租约过期的文件，commitOrCompleteLastBlock()方法会调用公有的重载方法 commitOrCompleteLastBlock()提交文件的最后一个数据块。

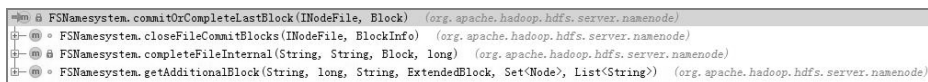


图 3-61 commitOrCompleteLastBlock()方法调用关系图

commitOrCompleteLastBlock()首先调用 BlockManager.commitBlock()方法提交数据块，将数据块的状态从 UnderConstruction 转变为 COMMITTED。如果这个数据块至少有一个以上的副本，则调用 BlockManager.completeBlock()将数据块状态转变为 COMPLETE。

```
public boolean commitOrCompleteLastBlock(BlockCollection bc,
    Block commitBlock) throws IOException {
    BlockInfo lastBlock = bc.getLastBlock();
    // 首先调用 commitBlock() 提交数据块
    final boolean b = commitBlock((BlockInfoUnderConstruction)lastBlock, commitBlock);
    // 如果当前数据块至少有一个以上的副本
    // 则调用 completeBlock() 将状态改为 COMPLETE
    if(countNodes(lastBlock).liveReplicas() >= minReplication)
        completeBlock(bc, bc.numBlocks()-1, false);
    return b;
}
```

BlockManager.commitBlock()的逻辑很简单，调用 BlockInfoUnderConstruction.commitBlock()将当前 BlockInfoUnderConstruction 的状态设置为 COMMITTED，之后用 Client 上传的数据块信息更新当前 BlockInfoUnderConstruction 的长度以及时间戳。

```
void commitBlock(Block block) throws IOException {
    // 将当前数据块状态更改为 COMMITTED
    blockUCState = BlockUCState.COMMITTED;
    // 更新当前数据块的长度以及时间戳
    this.set(getBlockId(), block.getNumBytes(), block.getGenerationStamp());
    // ...
}
```

BlockManager.completeBlock()调用 BlockInfoUnderConstruction.convertToCompleteBlock()方法将 BlockInfoUnderConstruction 对象转换为普通的 BlockInfo 对象，也就是隐式地将 BlockInfo 的状态变成了 COMPLETED，因为 BlockInfo.getBlockUCState()方法是始终返回 COMPLETED 状态的。更新了 BlockInfo 的状态后，就需要用这个新的 BlockInfo 对象去替代文件系统目录树中 INode.blocks 字段中保存的 BlockInfo 对象，以及 BlockManager.blocksMap 字段中保存的 BlockInfo 对象。

```
private BlockInfo completeBlock(final BlockCollection bc,
    final int blkIndex, boolean force) throws IOException {
    // 将 BlockInfoUnderConstruction 对象转换为普通的 BlockInfo 对象
    BlockInfo completeBlock = ucBlock.convertToCompleteBlock();
    // 更新 INode.blocks 中的数据块引用
    bc.setBlock(blkIndex, completeBlock);
    // 更新 BlockManager.blocksMap 中的数据块引用
    return blocksMap.replaceBlock(completeBlock);
}
```

4. 添加一个新的数据块

完成了上述所有操作之后，getAdditionalBlock()方法就可以进行分配新的数据块操作了。getAdditionalBlock()首先调用 createNewBlock()创建一个 Block 对象，并赋予一个新的时间戳。然后调用 saveAllocatedBlock()方法将 Block 对象转换为一个 BlockInfoUnderConstruction 对象，放入 INode.blocks 字段以及 BlockManager.blocksMap 字段中，最后调用 persistNewBlock()方法将操作记录在 editlog 中。完成了添加新的数据块操作之后，getAdditionalBlock()方法会调用 makeLocatedBlock()方法将新添加的数据块，以及分配的保存这个数据块副本的数据节点列表通过一个 LocatedBlock 对象返回给 Client。

```
LocatedBlock getAdditionalBlock(...) {
    // ...
    newBlock = createNewBlock();
    INodesInPath inodesInPath = INodesInPath.fromINode(pendingFile);
    saveAllocatedBlock(src, inodesInPath, newBlock, targets);
    persistNewBlock(src, pendingFile);
    offset = pendingFile.computeFileSize();

    getEditLog().logSync();
    // 返回 LocatedBlock 对象
    return makeLocatedBlock(newBlock, targets, offset);
    // ...
}
```

3.6.4 放弃数据块

当数据流管道创建失败时，Client 会调用 ClientProtocol.abandonBlock()放弃这个已经申请的数据块，之后重新申请数据块。在创建数据流管道时出错的 Datanode 将会加入 excludeNodes 中，在下次进行数据块的分配操作时，不再从这个 Datanode 分配数据块。

放弃数据块的操作是由 `FSNamesystem.abandonBlock()` 实现的，它的逻辑非常简单，由于错误是在数据流管道建立过程中发生的，`Datanode` 上并没有写入任何数据，所以 `abandonBlock()` 将从这个 HDFS 文件对应的 `INodeFile` 对象上删除这个数据块，并从 `BlocksMap` 中移出数据块信息。由于逻辑比较简单，这里就不再赘述了，感兴趣的读者请自行参考代码。

3.6.5 关闭文件

当 `Client` 完成了对文件的写操作后，会调用 `ClientProtocol.complete()` 方法关闭文件。关闭文件操作是由 `FSNamesystem.completeFile()` 和 `completeFileInternal()` 方法执行的，当成功地关闭文件之后，方法返回 `true`，否则返回 `false`。

`completeFileInternal()` 方法会检查文件租约，确认文件的倒数第二个数据块为 `COMPLETED` 状态。之后调用 `commitOrCompleteLastBlock()` 提交最后一个数据块，并在数据块副本数大于 1 时将该数据块的状态转换为 `COMPLETED`。最后调用 `finalizeINodeFileUnderConstruction()` 方法将文件由构建中状态转换为正常状态。

```
private boolean completeFileInternal(String src,
    String holder, Block last, long fileId) throws SafeModeException,
    UnresolvedLinkException, IOException {
    assert hasWriteLock();
    final INodeFile pendingFile;
    try {
        final INode inode;
        if (fileId == INodeId.GRANDFATHER_INODE_ID) {
            // 没有获得 fileId 时，只能通过 src 路径解析获取 inode 引用
            final INodesInPath iip = dir.getLastINodeInPath(src);
            inode = iip.getINode(0);
        } else {
            inode = dir.getINode(fileId);
            if (inode != null) src = inode.getFullPathName();
        }
        // 检查租约
        pendingFile = checkLease(src, holder, inode, fileId);
    } catch (LeaseExpiredException lee) {
        final INode inode = dir.getINode(src);
        if (inode != null
            && inode.isFile()
            && !inode.asFile().isUnderConstruction()) {
            // 租约过期，但是文件已经完成 complete 操作
            // 这种情况可能发生在 complete() 请求已经执行，但是响应没有到达 Client
            // Client 发起了重试操作
            final Block realLastBlock = inode.asFile().getLastBlock();
            if (Block.matchingIdAndGenStamp(last, realLastBlock)) {
                return true;
            }
        }
    }
}
```



```

        throw lee;
    }
    // 检查文件的倒数第二个数据块的状态是否为 COMPLETED
    if (!checkFileProgress(pendingFile, false)) {
        return false;
    }

    // 提交最后一个数据块, 如果该数据块的副本数大于1, 则将状态转换为 COMPLETED
    commitOrCompleteLastBlock(pendingFile, last);

    // 检查文件的所有数据块是否都为 COMPLETED 状态
    if (!checkFileProgress(pendingFile, true)) {
        return false;
    }

    // 完成 INode 写操作, 将 INode 转换为正常状态
    finalizeINodeFileUnderConstruction(src, pendingFile,
        Snapshot.CURRENT_STATE_ID);
    return true;
}

```

`commitOrCompleteLastBlock()`方法的实现我们在上一节中已经介绍过了, 下面看一下 `finalizeINodeFileUnderConstruction()`方法的实现。`finalizeINodeFileUnderConstruction()`首先从租约管理器中删除文件的租约, 调用 `INodeFile.toCompleteFile()`从 `INodeFile` 中删除 `UnderConstructionsFeature`, 也就是将文件从构建中状态转换为正常状态。之后调用 `closeFile()`在 `editlog` 中记录关闭操作, 最后调用 `blockManager.checkReplication()`确认文件拥有的所有数据块的副本数足够, 如果不够则加入 `BlockManager.neededReplications` 队列中。

```

private void finalizeINodeFileUnderConstruction(String src,
    INodeFile pendingFile, int latestSnapshot) throws IOException,
    UnresolvedLinkException {
    assert hasWriteLock();

    FileUnderConstructionFeature uc = pendingFile.getFileUnderConstructionFeature();
    Preconditions.checkArgument(uc != null);
    leaseManager.removeLease(uc.getClientName(), src);

    pendingFile.recordModification(latestSnapshot);

    // 删除 underconstructionFeature
    final INodeFile newFile = pendingFile.toCompleteFile(now());

    waitForLoadingFSImage();
    closeFile(src, newFile);

    // ...
    blockManager.checkReplication(newFile);
}

```

至此，FSNamesystem 中与文件写操作相关的方法就分析完毕了。总结：Client 首先调用 ClientProtocol.create() 方法创建一个新的 HDFS 文件，然后调用 ClientProtocol.addBlock() 方法获取一个新的数据块的位置信息，或者通过调用 ClientProtocol.append() 追加写接口获取已有 HDFS 文件中最后一个未写满数据块的位置信息。有了数据块的位置信息，Client 就可以建立数据流管道并进行写操作了，当 Client 完成一个数据块的写操作后，会调用 addBlock() 申请新的数据块并提交上一个完成写操作的数据块。Datanode 成功写入了数据块的副本之后，会调用 DatanodeProtocol.blockReceivedAndDeleted() 向 Namenode 汇报。当 Client 完成了整个文件的写操作之后，会调用 ClientProtocol.complete() 方法向 Namenode 提交最后一个数据块，将文件从构建中状态改为正常状态并释放租约。

3.7 Namenode 的启动和停止

了解了 Namenode 中各个模块的实现，我们就可以学习 Namenode 的启动了。Hadoop 2.X 中 Namenode 的启动比 Hadoop 1.X 中复杂很多，这是因为 Hadoop 2.X 中引入了 HA 机制。所以本章除了介绍 Namenode 的启动流程和安全模式外，还会涉及 Namenode 的 HA 机制。

3.7.1 安全模式

安全模式是 Namenode 的一种状态，处于安全模式中的 Namenode 不接受任何对于命名空间的修改操作，同时也不触发任何复制和删除数据块的操作。

Namenode 启动时会自动进入安全模式状态，用户可以通过 “dfsadmin -safemode value” 命令来操作安全模式，这个命令在底层是由 DFSClient 调用 ClientProtocol.setSafeMode(SafeModeAction.SAFEMODE_ENTER,false) 方法实现的。

Namenode 启动时会首先加载命名空间镜像 (fsimage) 并且合并编辑日志 (editslog)，完成这些操作后 Namenode 的第一关系 (文件系统目录树) 也就建立起来了。之后 Namenode 就需要接受 Datanode 的块汇报 (blockReport) 以获得数据块的存储信息，也就是建立第二关系 (数据块与存储这个数据块副本的 Datanode 的对应关系)。这些操作都是 Namenode 在安全模式中进行的，只有当 Namenode 收集到的阈值比例满足最低副本系数的数据块时才可以离开安全模式。最低副本系数指的是一个数据块应该拥有的最少的副本数量，它是由配置项 dfs.namenode.replication.min 配置的，默认值是 1。这里的阈值比例指的是已经收集到的满足最低副本系数的数据块数量与 HDFS 文件系统中所有数据块的比例，文件系统中所有数据块的数量在第一关系建立时就可以获得，阈值比例则是由配置项 dfs.safemode.threshold.pct 配置的，默认是 0.999。当 Namenode 发现已经满足了阈值比例后，会延迟一段时间退出安全模式，目的是等待那些还没有进行块汇报的数据节点进行块汇报，这个时间是由配置项 dfs.safemode.extension 配置的，默认是 30 秒。之后 Namenode 就可以顺利地退出安全模式了。

这里要注意的是，如果客户端通过远程接口 ClientProtocol.setSafeMode() 进入安全模式，是不可以自动退出安全模式的，必须由客户端手动调用远程接口 setSafeMode(SafeModeAction.

SAFEMODE_LEAVE,false)退出安全模式。

表 3-4 总结了上面提到的安全模式相关配置项。

表 3-4 安全模式相关配置项

配置名	类型	默认值	描述
dfs.namenode.replication.min	int	1	数据块最低副本系数，也用于写操作时判断是否可以 complete 一个数据块
dfs.safemode.threshold.pct	float	0.999	离开安全模式时，系统需要满足的阈值比例。也就是满足最低副本系数的数据块与系统内所有数据块的比例
dfs.safemode.extension	int	30000	安全模式等待时间，也就是满足了最低副本系数之后，离开安全模式的时间，用于等待剩余的数据节点上报数据块

注意：如果 threshold 设置为 0，则 Namenode 启动时并不会进入安全模式。如果 threshold 设置为 1，则 Namenode 需要等待所有数据块上报之后才能退出安全模式。如果 threshold 设置大于 1，则 Namenode 无法自动退出安全模式。同时需要注意的是，任何时候都可以通过手动方式退出安全模式。

1. SafeModeInfo

如图 3-62 所示，在 HDFS 源码中使用 SafeMode 接口抽象所有与安全模式相关的操作，SafeMode 接口是由 FSNamesystem 类实现的。

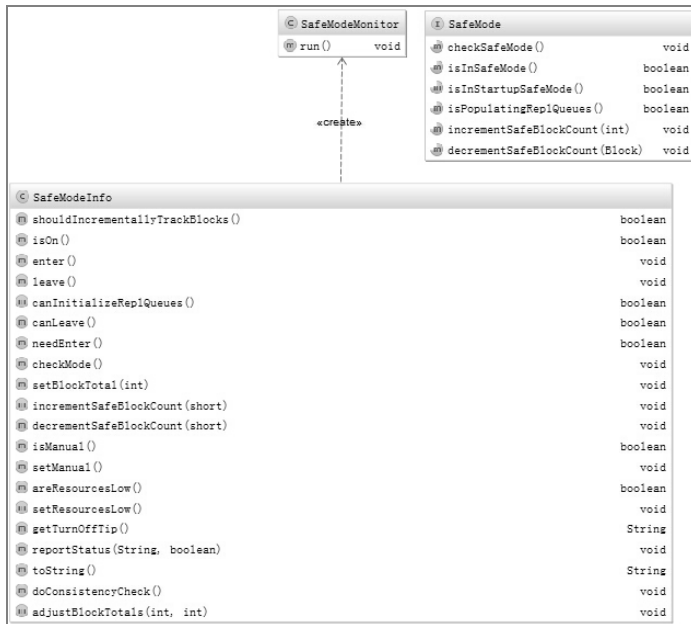



图 3-62 SafeModeInfo 类结构图

在 FSNamesystem 中则使用内部类 SafeModeInfo 记录安全模式的配置信息、Namenode 收集到的数据块信息（Datanode 汇报的数据块）以及 Namenode 当前的安全模式状态（Namenode 是否处于安全模式中）。Namenode 会在重建第一关系和第二关系时调用 SafeModeInfo 提供的方法更新 SafeModeInfo 中记录的数据块信息，而 SafeModeInfo 类会启动一个 SafeModeMonitor 线程监控这些数据块信息，如果符合离开安全模式的条件，SafeModeMonitor 线程会改变 SafeModeInfo 中记录的 Namenode 的安全模式状态。

（1）字段

图 3-63 给出了 SafeModeInfo 定义的所有字段。threshold 字段用于保存离开安全模式的阈值比例，safeReplication 字段用于保存数据块最低副本系数，extension 字段用于保存离开安全模式等待时间，这三个字段对应于表 3-4 中介绍的三个安全模式配置项。blockSafe 和 blockTotal 这两个字段分别用于记录已经满足最小副本数的数据块数量，以及 HDFS 文件系统中所有数据块的数量。reached 字段则用于记录安全模式的状态，reached=0 表示当前线程处于安全模式；reached>0 则表示阈值系数已经满足了，但是安全模式还在等待退出时间；reached<0 表示已经退出了安全模式。



SafeModeInfo	
threshold	double
datanodeThreshold	int
extension	int
safeReplication	int
replQueueThreshold	double
reached	long
blockTotal	int
blockSafe	int
blockThreshold	int
blockReplQueueThreshold	int
lastStatusReport	long
resourcesLow	boolean
shouldIncrementallyTrackBlocks	boolean
awaitingReportedBlocksCounter	Counter

图 3-63 SafeModeInfo 字段

在 SafeModeInfo 中对于安全模式的管理主要是通过 reached 字段完成的，reached 字段的设置是通过调用 enter() 和 leave() 这两个方法实现的。调用 enter() 方法会进入安全模式，设置 reached 字段为 0；调用 leave() 方法会退出安全模式，设置 reached 字段为 -1，并且释放 safeModeInfo 对象。而 isOn() 方法则用于判断当前 Namenode 是否处于安全模式中，它直接判断 reached 字段的值即可。

```
private void enter() {
    this.reached = 0;
}

private synchronized void leave() {
    // 判断是否可以开启复制以及删除数据块的操作
    if (!isPopulatingReplQueues() && shouldPopulateReplQueues()) {
        initializeReplQueues();
    }
}
```

```
// ...
reached = -1; // 设置 reached == -1, 表明已经离开了安全模式
safeMode = null; // 置空 FSNamesystem.safeModeInfo 对象
// ...
}

private synchronized boolean isOn() {
    doConsistencyCheck();
    return this.reached >= 0;
}
```

对于 `blockTotal` 和 `blockSafe` 字段则是在 Namenode 建立第一关系和第二关系时设置的。我们知道当 Namenode 成功地加载了 `fsimage` 以及 `editlog` 之后, 第一关系也就建立了, 这时就可以通过遍历文件系统目录树中所有文件拥有的数据块而计算出 `blockTotal` 字段, 然后通过 `SafeModeInfo.setBlockTotal()` 方法设置 `blockTotal` 字段。

```
private synchronized void setBlockTotal(int total) {
    this.blockTotal = total;
    this.blockThreshold = (int) (blockTotal * threshold);
    this.blockReplQueueThreshold =
        (int) (blockTotal * replQueueThreshold);
    if (haEnabled) {
        this.shouldIncrementallyTrackBlocks = true;
    }
    if (blockSafe < 0)
        this.blockSafe = 0;
    checkMode();
}
```

而对于 `blockSafe` 字段则是在数据节点向名字节点进行块汇报, 也就是第二关系建立的过程中不断修改的。当 Namenode 处理数据节点的第一个块汇报时, 或者当 Datanode 进行普通块汇报或者增量块汇报时, Namenode 会调用 `BlockManager.addStoredBlockImmediate()` 以及 `addStoredBlock()` 方法在第二关系中加入这个副本, 这两个方法都会调用 `SafeModeInfo.incrementSafeBlockCount()` 方法增加 `blockSafe` 字段的值。

```
private synchronized void incrementSafeBlockCount(short replication) {
    if (replication == safeReplication) { // 达到了最低副本系数, 则增加 blockSafe 字段的值
        this.blockSafe++;
        checkMode();
    }
}
```

同理, 当数据节点向名字节点汇报数据块的一个副本删除时, Namenode 会调用 `BlockManager.removeStoredBlock()` 从第二关系中移除这个副本, 这个方法还会调用 `SafeModeInfo.decrementSafeBlockCount()` 修改 `blockSafe` 字段的值。

```
private synchronized void decrementSafeBlockCount(short replication) {
    if (replication == safeReplication-1) { // 低于最低副本系数, 则减小 blockSafe 字段的值
        this.blockSafe--;
    }
}
```

Hadoop 2.X HDFS 源码剖析

```
        checkMode();
    }
}
```

我们注意到上面的三个方法在方法体的最后都会调用 `checkMode()` 方法，`checkMode()` 方法用于检查安全模式的状态，主要是判断阈值系数是否满足，以及满足阈值系数之后是否等待了足够的时间。对于能否离开安全模式的两个条件判断，`checkMode()` 方法会调用 `needEnter()` 方法判断阈值系数是否满足。对于等待时间，`checkMode()` 则会启动一个 `SafeModeMonitor` 线程，每隔 1 秒检查一次等待时间是否满足，如果满足则直接调用 `leave()` 方法退出安全模式。

```
private void checkMode() {
    // 如果 smmthread 线程没有启动
    // 并且调用 needEnter() 判断当前状态是否需要进入安全模式
    if (smmthread == null && needEnter()) {
        // 调用 enter() 方法进入安全模式，设置 reached=0
        enter();
        // 检查是否可以产生复制以及删除副本的队列
        // 这里判断的条件是当前 Namenode 是 Active 并且满足离开安全模式的条件
        if (canInitializeReplQueues() && !isPopulatingReplQueues()
            && !haEnabled) {
            initializeReplQueues();
        }
        return;
    }

    if (!isOn() || // 如果已经满足离开条件，reached<0
        extension <= 0 || threshold <= 0) { // 不需要等待，立即离开安全模式
        this.leave(); // 离开安全模式，设置 reached=-1
        return;
    }
    if (reached > 0) { // 已经不是第一次满足阈值系数，且还需要等待，则返回
        return;
    }
    // 创建 SafeModeMonitor 对象并启动，判断是否等待了足够的时间
    reached = now();
    if (smmthread == null) {
        smmthread = new Daemon(new SafeModeMonitor());
        smmthread.start();
    }

    // 再次检查是否满足了开启副本复制以及删除的条件
    if (canInitializeReplQueues() && !isPopulatingReplQueues() && !haEnabled) {
        initializeReplQueues();
    }
}

// 判断是否进入安全模式，在阈值系数不满足的情况下进入安全模式等待
private boolean needEnter() {
```

```

return (threshold != 0 && blockSafe < blockThreshold) ||
    (datanodeThreshold != 0 && getNumLiveDataNodes() < datanodeThreshold) ||
    (!nameNodeHasResourcesAvailable());
}

```

(2) SafeModeMonitor

SafeModeMonitor 的实现也比较简单，它循环调用 SafeModeInfo.canLeave()方法判断是否满足上面提到的两个条件，然后离开安全模式。代码如下：

```

class SafeModeMonitor implements Runnable {
    private static final long recheckInterval = 1000;    // 检查间隔时间为 1 秒

    public void run() {
        while (fsRunning) {
            writeLock();
            try {
                if (safeMode.canLeave()) {    //调用 canLeave() 方法判断能否离开安全模式
                    safeMode.leave();        // 满足条件则直接离开安全模式
                    smmthread = null;
                    break;
                }
            } finally {
                writeUnlock();
            }

            try {
                Thread.sleep(recheckInterval);    // 等待 1 秒
            } catch (InterruptedException ie) {
            }
        }
    }
}

```

canLeave()方法的实现也非常简单，如果 reached==0 也就是正处于安全模式状态时，则返回 false。如果已经满足了副本阈值，但是没有等待足够的时间，则不能退出，返回 false。如果调用 needEnter()判断不满足副本阈值，则返回 false。最后，也就是当前 Namenode 满足了退出安全模式的两个条件，则返回 true 直接退出。

```

private synchronized boolean canLeave() {
    if (reached == 0) {
        return false;
    }
    if (now() - reached < extension) {
        return false;
    }
    if (needEnter()) {
        return false;
    }
}

```

```
    return true;
}
```

2. 开启复制和删除数据块功能

我们知道在安全模式中，Namenode 是不会开启复制数据块和删除数据块功能的。但是在如下两种情况下，Namenode 会调用 `initializeReplQueues()` 方法启动复制和删除操作。

- 在 `checkMode()` 方法中会调用 `canInitializeReplQueues()` 方法判断修改了 `blockSafe` 数量的当前 Namenode 是否满足重新开启复制和删除操作的条件，如果满足则调用 `initializeReplQueues()` 方法启动复制和删除数据块功能。
- 在 `leave()` 方法中，也就是 Namenode 离开安全模式时，由于已经离开了安全模式，也就没有必要关闭复制和删除操作了，所以这时会调用 `initializeReplQueues()` 方法恢复复制和删除数据块的功能。

`canInitializeReplQueues()` 方法用于判断 Namenode 是否达到了重启复制和删除数据块功能的条件。这里的判断条件是当前 Namenode 已经接收的 `blockSafe` 的数据块数量是否达到了恢复复制和删除数据块功能的数据块数量 (`blockReplQueueThreshold`)，`blockReplQueueThreshold` 的大小与离开安全模式所需要的数据块数量 (`blockThreshold`) 相同。同时要注意，只有 Active Namenode 才能调用 `canInitializeReplQueues()` 方法。

```
private synchronized boolean canInitializeReplQueues() {
    return shouldPopulateReplQueues()
        && blockSafe >= blockReplQueueThreshold;
}
```

`initializeReplQueues()` 则用于重新启动复制和删除数据块功能，它会调用 `BlockManager.initializeReplQueues()` 方法开启复制和删除队列。这部分代码我们在数据块管理小节中已经介绍过了，这里不再详细介绍。

```
private void initializeReplQueues() {
    blockManager.processMisReplicatedBlocks(); // 启动复制和删除队列
    initializedReplQueues = true;
}
```

3. 安全模式管理

我们知道客户端对于安全模式的管理是通过调用 `ClientProtocol.setSafeMode()` 方法实现的，这个方法可以通过传入参数的不同实现进入安全模式、离开安全模式，以及判断是否在安全模式中等功能。`ClientProtocol.setSafeMode()` 方法是由 `FSNamesystem.setSafeMode()` 方法响应的，这个方法的实现也非常简单，就是判断传入参数然后调用对应的方法即可。

```
boolean setSafeMode(SafeModeAction action) throws IOException {
    if (action != SafeModeAction.SAFEMODE_GET) {
        switch(action) {
            case SAFEMODE_LEAVE: // 离开安全模式
                leaveSafeMode();
                break;
        }
    }
}
```



```

        case SAFEMODE_ENTER: // 进入安全模式
            enterSafeMode(false);
            break;
        default:
    }
}
return isInSafeMode(); // 返回是否处于安全模式中
}

```

`enterSafeMode()`方法会构造 `SafeModeInfo` 对象,并将这个对象保存在 `FSNamesystem.safeMode` 字段中。注意,如果是用户手动开启安全模式的,则必须手动关闭安全模式,无法等待安全模式自动关闭。`enterSafeMode()`方法会判断,如果是用户手动打开了安全模式,则调用 `SafeModeInfo.setManual()`方法将 `SafeModeInfo.extension` 字段设置为无限大(`Integer` 的最大值),这样即使 Namenode 满足了阈值系数也是无法离开安全模式的,只有等待管理员手动关闭安全模式。

```

void enterSafeMode(boolean resourcesLow) throws IOException {
    writeLock();
    try {
        // ...
        if (!isInSafeMode()) {
            safeMode = new SafeModeInfo(resourcesLow); // 构造 SafeModeInfo 对象
            return;
        }
        if (resourcesLow) {
            safeMode.setResourcesLow();
        } else {
            safeMode.setManual();
        }
        // ...
    } finally {
        writeUnlock();
    }
}

```

而 `leaveSafeMode()` 和 `isInSafeMode()` 方法的实现都比较简单,它们直接调用了 `SafeModeInfo.leave()`方法以及 `SafeModeInfo.isOn()`方法。

```

void leaveSafeMode() {
    writeLock();
    try {
        if (!isInSafeMode()) {
            return;
        }
        safeMode.leave();
    } finally {
        writeUnlock();
    }
}

```

```
public boolean isInSafeMode() {
    SafeModeInfo safeMode = this.safeMode;
    if (safeMode == null)
        return false;
    return safeMode.isOn();
}
```

3.7.2 HDFS High Availability

在 Hadoop2.X 之前，Namenode 是 HDFS 集群中可能发生单点故障的节点，每个 HDFS 集群中只有一个 Namenode，一旦这个节点不可用，则整个 HDFS 集群将处于不可用状态。

HDFS 的高可用（High Availability, HA）方案就是为了解决上述问题而产生的，在 HA HDFS 集群中会同时运行两个 Namenode，一个作为活动的（Active）Namenode，一个作为备份的（Standby）Namenode。备份 Namenode 的命名空间与活动 Namenode 是实时同步的，所以当活动 Namenode 发生故障而停止服务时，备份 Namenode 可以立即切换为活动状态，而不影响 HDFS 集群的服务。本节就介绍 Hadoop 2.6 版本中 HA 机制的实现。

1. HA 架构

在一个 HA 集群中，会配置两个独立的 Namenode。在任意时刻，只有一个节点会作为活动的节点，另一个节点则处于备份状态。活动的 Namenode 负责执行所有修改命名空间以及删除备份数据块的操作，而备份的 Namenode 则执行同步操作以保持与活动节点命名空间的一致性。

如图 3-64 所示，为了使备份节点与活动节点的状态能够同步一致，两个节点都需要与一组独立运行的节点（JournalNodes, JNS）通信。当 Active Namenode 执行了修改命名空间的操作时，它会定期将执行的操作记录在 editlog 中，并写入 JNS 的多数节点中。而 Standby Namenode 会一直监听 JNS 上 editlog 的变化，如果发现 editlog 有改动，Standby Namenode 就会读取 editlog 并与当前的命名空间合并。当发生了错误切换时，Standby 节点会先保证已经从 JNS 上读取了所有的 editlog 并与命名空间合并，然后才会从 Standby 状态切换为 Active 状态。通过这种机制，保证了 Active Namenode 与 Standby Namenode 之间命名空间状态的一致性，也就是第一关系链的一致性。

为了使错误切换能够很快地执行完毕，就需要保证 Standby 节点也保存了实时的数据块存储信息，也就是第二关系链。这样发生错误切换时，Standby 节点就不需要等待所有的数据节点进行全量块汇报，而可以直接切换为 Active 状态。为了实现这个机制，Datanode 会同时向这两个 Namenode 发送心跳以及块汇报信息。这样 Active Namenode 和 Standby Namenode 的元数据就完全同步了，一旦发生故障，就可以马上切换，也就是热备。这里需要注意的是，Standby Namenode 只会更新数据块的存储信息，并不会向 Namenode 发送复制或者删除数据块的指令，这些指令只能由 Active Namenode 发送。

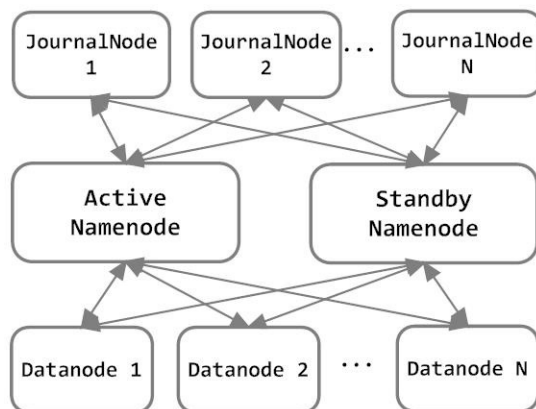


图 3-64 HDFS HA 架构示意图

在 HA 架构中有一个非常重要的问题，就是需要保证同一时刻只有一个处于 Active 状态的 Namenode，否则就会出现两个 Namenode 同时修改命名空间的问题，也就是脑裂（split-brain）。脑裂的 HDFS 集群很有可能造成数据块的丢失，以及向 Datanode 下发错误指令等异常情况。为了预防脑裂的情况，HDFS 提供了三个级别的隔离（fencing）机制。

- 共享存储隔离：同一时间只允许一个 Namenode 向 JournalNodes 写入 editlog 数据。
- 客户端隔离：同一时间只允许一个 Namenode 响应客户端请求。
- Datanode 隔离：同一时间只允许一个 Namenode 向 Datanode 下发名字节点指令，例如删除、复制数据块指令等。

2. HA 配置

HDFS 管理员要开启 HA 功能是需要配置文件中添加 HA 配置的，本节就介绍 HA 配置。HDFS 的 HA 配置是向后兼容的，它允许将单 Namenode 的集群配置成 HA 集群。新的配置方式确保了集群中所有节点的配置文件都是相同的，不会因为节点的不同而分配不同的配置文件。

在 HA 配置中使用 nameservice ID 来唯一标识一个 HDFS 实例，这个 HDFS 实例可能包括多个 Namenode。此外，在 HA 配置中添加了 namenode ID 用于标识集群中唯一的 Namenode。为了确保集群中所有的 Namenode 都拥有相同的配置文件，相关的配置参数都会以 nameservice ID 和 namenode ID 作为后缀。

在配置 HA Namenodes 时，需要增加一些配置项到 hdfs-site.xml 文件中。这些配置项的顺序并不重要，但 dfs.federation.nameservices 和 dfs.ha.namenodes.[nameservice ID] 两个配置项的值将决定下面配置的 Key 值，所以我们会优先确定这两个配置项的值。

在 hdfs-site.xml 配置文件中与 HA 相关的配置项如下。

- **dfs.federation.nameservices**：这个配置项用于定义当前集群所拥有的所有命名空间服务的逻辑名，以逗号分隔。我们知道引入了 federation 机制之后，一个 HDFS 集群可以拥有多个命名空间的服务，对于每个独立的命名空间服务都会拥有自己独立的

Active Namenode 和 Standby Namenode。如下例所示，当前 HDFS 集群只有一个逻辑名为“mycluster”的命名空间服务。

```
<property>
<name>dfs.federation.nameservices</name>
<value>mycluster</value>
</property>
```

- **dfs.ha.namenodes.[nameservice ID]**: 这个配置项用于定义在[nameservice ID]这个命名空间服务下的两个 Namenode（分别作为 Active 和 Standby 节点运行）的逻辑名。之后针对每个 Namenode 的逻辑名，配置其对应的 dfs.namenode.rpc-address.[nameservice ID].[namenode ID] 配置项来指定该 Namenode 的 RPC 服务地址。如下例所示，对于我们定义的“mycluster”这个命名空间服务，定义 namenode ID 为“nn1”和“nn2”的两个 Namenode。

```
<property>
<name>dfs.ha.namenodes.mycluster</name>
<value>nn1,nn2</value>
</property>
```

- **dfs.namenode.shared.edits.dir**: 这个配置项用于定义存放 editlog 的共享存储位置。尽管这里需要配置多个 JNS 的位置，但是路径只能有一个，并且这个路径对于两个 Namenode 都必须是可读写的，且必须是绝对路径。如下例所示，定义了命名空间服务“mycluster”的共享存储 JNS，JNS 分别为“node1.example.com”、“node2.example.com”和“node3.example.com”。

```
<property>
<name>dfs.namenode.shared.edits.dir</name>
<value>qjournal://node1.example.com:8485;node2.example.com:8485;node3.example.com:8485/mycluster</value>
</property>
```

- **dfs.client.failover.proxy.provider.[nameservice ID]**: 这个配置项用于定义具体的 failover proxy provider 类，也就是当客户端发现原来的 Active Namenode 无法连接时，该如何确定并连接新的 Active Namenode。目前 Hadoop 只定义了一个策略类 Configured FailoverProxyProvider，实现逻辑是如果发现无法发送 RPC 请求时，下一次把 RPC 请求发送给另一个 Namenode 的 proxy。

```
<property>
<name>dfs.client.failover.proxy.provider.mycluster</name>
<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>
</property>
```

- **dfs.ha.fencing.methods**: 这个配置项定义了用于停止故障的 Active Namenode 的脚本或 Java 类列表。在 HA 模式下，需要保证任何时候只有一个 Namenode 处于 Active 状态。我们知道 JNS 已经保证了只有一个 Namenode 可以写入 editlog，但是依旧可能发生已经切换的 Active 节点还在响应来自客户端的请求，直到这个节点向 JNS 写入数据时被终止。因此在故障转移期间，在启动 Standby 节点前，首先要确保 Active 节点处于等待状态，或者进程被中止。为了达到这个目的，至少需要配置一个强行

中止 Active 节点的方法，或者一个终止方法的列表，HDFS 按顺序执行列表中的方法，直到其中一个返回成功。HDFS 提供了两种 fencing 方式，即 shell 和 sshfence。sshfence 方式通过 SSH 连接活动的 Namenode，杀掉进程，为了实现 SSH 登录杀掉进程，还需要配置免密码登录的 SSH 密钥信息。shell 方式执行任意的 shell 命令来终止活动的 Namenode。如果要定义自己的 fencing 方法，可以参考 `org.apache.hadoop.ha.NodeFencer` 类。

3. 管理命令

当完成了 HA 配置后，就可以启动 Standby Namenode 和 Active Namenode 了，Namenode 启动后的初始状态都是 Standby。这时就可以通过 HDFS 提供的 DFSHAAdmin 命令来管理 HA 集群了。DFSHAAdmin 命令的格式如下：

```
Usage: DFSHAAdmin [-ns <nameserviceId>]
      [-transitionToActive <serviceId>]
      [-transitionToStandby <serviceId>]
      [-failover [--forcefence] [--forceactive] <serviceId><serviceId>]
      [-getServiceState <serviceId>]
      [-checkHealth <serviceId>]
      [-help <command>]
```

下面我们依次介绍每个参数的作用。

- **transitionToActive&transitionToStandby**: 这个参数用于把指定的 Namenode 转换成 Active 或 Standby 状态，但这个命令不会试图去停止 Active Namenode，所以尽量少使用这个参数，可以使用“`hdfs haadmin -failover`”来代替。
- **failover**: 这个参数用于在两个 Namenode 之间进行故障转移。如果第一个 Namenode 处于 Standby 状态，这个命令会直接将第二个 Namenode 转换成 Active 状态；如果第一个 Namenode 处于 Active 状态，则它会尝试将第一个 Namenode 转换为 Standby 状态，同时从配置项 `dfs.ha.fencing.methods` 定义的方法列表中的第一个方法开始调用，直到有方法成功地停止了 Active Namenode，它才会把第二个 Namenode 转换成 Active 状态。如果上述方法列表中没有一个方法成功，则第二个 Namenode 不能被转换为 Active 状态，并且会返回一个错误。
- **getServiceState**: 这个参数用于获取指定 Namenode 的状态。这个方法会连接到指定的 Namenode 去确定它当前的状态，并将它的状态（Active 或者 Standby）输出到标准输出。
- **checkHealth**: 这个参数用于检测指定 Namenode 的健康状态。它会连接到指定的 Namenode 去检查它的健康状态，Namenode 能够对自身进行一些诊断，包括检查内部服务是否正常运行。如果 Namenode 运行正常，方法将返回 0；否则返回非 0 值，这个命令主要用于监测。

4. 源码实现

HDFS 提供了两种 HA 状态切换方式：一种是管理员手动通过命令执行状态切换；另一

种是自动状态切换机制触发状态切换。手动切换方式是管理员通过执行 HA 管理命令（请参考管理命令小节）触发切换操作，底层是由客户端调用 HAAdmin 类提供的对应方法实现的。而对于自动状态切换机制，则是由 ZKFailoverController 控制切换流程。无论 HAAdmin 还是 ZKFailoverController，都调用了 RPC 接口 HServiceProtocol 向 Namenode 发送 HA 请求，请求到达 Namenode 后，会由 NameNodeRpcServer 类响应。

在 HA 实现中还有一个非常重要的部分就是 Active Namenode 和 Standby Namenode 之间如何共享 editlog 日志文件。Active Namenode 会将日志文件写到共享存储上，Standby Namenode 会实时地从共享存储读取 editlog 文件，然后合并到 Standby Namenode 的命名空间中，这样一旦 Active Namenode 发生错误，Standby Namenode 可以立即切换为 Active 状态。Hadoop 2.6 提供了 QJM（Quorum Journal Manager）方案来解决 HA 共享存储的问题。

本节就介绍 HServiceProtocol、HAAdmin 以及 HServiceProtocol 的代码实现，同时介绍 QJM 方案的实现机制。

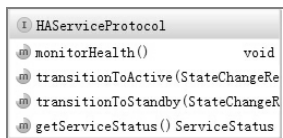


图 3-65 HServiceProtocol 接口定义

（1）HServiceProtocol

HDFS 的 HA 管理命令底层都是由 Client 调用远程 RPC 接口 HServiceProtocol 实现的，HServiceProtocol 接口抽象了所有与 HA 管理相关的方法，如图 3-65 所示。

HServiceProtocol 中定义的方法是在 NameNodeRpcServer 类中实现的，我们依次看一下接口中定义的 4 个方法是如何实现的。

monitorHealth()

monitorHealth()方法用于检查 Namenode 服务的状态，如果服务出现错误，则抛出异常，这时就可能触发一个 failover 操作。NameNodeRpcServer.monitorHealth() 会调用 Namenode.monitorHealth()方法执行检查操作，monitorHealth()方法会检查 Namenode 的磁盘空间是否足够可用，在不可用时会抛出 HealthCheckFailedException 异常。这个方法的实现比较简单，这里就不贴出代码了。

transitionToActive()

transitionToActive()方法用于将当前 Namenode 转换为 Active 状态。如下代码所示，transitionToActive()方法会依次调用 Namenode.checkHaStateChange()检查这次状态切换是否合法，之后调用 Namenode.transitionToActive()方法触发当前 Namenode 的状态转移。

checkHaStateChange()方法会判断由 Client 发起的切换请求或者自动异常转移流程触发的切换请求与配置文件是否匹配，如果不匹配则抛出 AccessControlException 异常。

transitionToActive()方法则直接调用 HState.setState()方法将当前 Namenode 的状态转换为 Active。HState 用于描述当前 Namenode 所处的 HA 状态，它有两个子类，即 ActiveState 和 StandbyState，分别对应于处于 Active 状态的 Namenode 以及处于 Standby 状态的 Namenode。

```
synchronized void transitionToActive()
```

```

        throws ServiceFailedException, AccessControlException {
    namesystem.checkSuperuserPrivilege();
    if (!haEnabled) {
        throw new ServiceFailedException("HA for namenode is not enabled");
    }
    state.setState(haContext, ACTIVE_STATE);
}

```

需要注意的是, `Namenode.state` 可能是 `ActiveState`, 也可能是 `StandbyState`。对于 `Namenode` 当前所处的不同状态, `HAState.setState()` 会调用不同的重写 (overload) 方法。如下代码所示, `StandbyState` 和 `ActiveState` 两个类都重写了父类 `HAState` 的 `setState()` 方法。

```

public class StandbyState extends HAState {
    public void setState(HAContext context, HAState s) throws ServiceFailedException {
        // 对于 Standby 节点, 只有转换为 Active 状态时, 才执行转换操作
        if (s == NameNode.ACTIVE_STATE) {
            setStateInternal(context, s);
            return;
        }
        super.setState(context, s);
    }
}

public class ActiveState extends HAState {
    public void setState(HAContext context, HAState s) throws ServiceFailedException {
        // 对于 Active 节点, 只有转换为 Standby 状态时, 才执行转换操作
        if (s == NameNode.STANDBY_STATE) {
            setStateInternal(context, s);
            return;
        }
        super.setState(context, s);
    }
}

```

可以看到 `StandbyState.setState()` 和 `ActiveState.setState()` 的逻辑是完全一致的, 不同的只是在状态判断一行代码上, 之后这两个方法都会调用 `setStateInternal()` 执行内部的状态转移流程。`setStateInternal()` 方法首先调用 `prepareToExitState()` 取消 `Standby Namenode` 上正在进行的检查点 (checkpoint) 操作, 需要注意的是检查点操作只在 `Standby` 节点上执行。之后 `setStateInternal()` 会调用 `exitState()` 方法停止当前 `Active` 或者 `Standby` 节点的服务。对于 `Standby` 节点, `exitState()` 方法会停止负责检查点操作的 `standbyCheckpoint` 线程, 以及追踪 `editlog` 更新的 `editLogTailer` 线程。而对于 `Active` 节点, `exitState()` 方法则将 `leaseManager`、`cacheManager`、`blockManager` 等管理类服务全部停止。完成 `exitState()` 调用后, `setStateInternal()` 会调用 `context.setState()` 方法将 `NameNodeHAContext` 中保存的 `Namenode` 状态设置为指定状态。最后 `setStateInternal()` 调用 `HAState.enterState()` 方法启动指定状态的所有服务。至此, `Namenode` 原有状态的所有服务已经停止, 切换后状态的服务也都顺利启动, 内部的状态切换完成。`setStateInternal()` 方法的代码如下:

```

protected final void setStateInternal(final HAContext context, final HAState s)

```

```
        throws ServiceFailedException {
    prepareToExitState(context);
    s.prepareToEnterState(context);
    context.writeLock();
    try {
        exitState(context);
        context.setState(s);
        s.enterState(context);
    } finally {
        context.writeUnlock();
    }
}
```

这里我们模拟 Standby -> Active 的切换流程，追踪 `setStateInternal()` 的代码实现，`setStateInternal()` 会调用 `exitState()` 方法使得 Namenode 退出当前 Standby 状态。`exitState()` 会首先调用 `NameNodeHAContext.stopStandbyService()` 方法停止 Standby Namenode 的所有服务，`stopStandbyService()` 调用是由 `FSNamesystem.stopStandbyService()` 方法响应的。成功停止了 Standby Namenode 的服务后，`setStateInternal()` 方法会调用 `ActiveState.enterState(context)` 启动 Active Namenode 的所有服务，`enterState(context)` 最终是由 `FSNamesystem.startActiveServices()` 方法响应的。`FSNamesystem` 的 `stopStandbyService()` 以及 `startActiveServices()` 方法的实现请参考下面的代码。

```
void stopStandbyServices() throws IOException {
    // 关闭负责进行检查点操作的 standbyCheckpoint 线程
    if (standbyCheckpoint != null) {
        standbyCheckpoint.stop();
    }
    // 关闭监控 editlog 内容的 editLogTailer 线程
    if (editLogTailer != null) {
        editLogTailer.stop();
    }
    // 关闭当前的 editlog
    if (dir != null && getFSImage() != null && getFSImage().editLog != null) {
        getFSImage().editLog.close();
    }
}

void startActiveServices() throws IOException {
    startingActiveService = true;
    writeLock();
    try {
        // 开启新的 editlog 输出流
        FSEditLog editLog = getFSImage().getEditLog();

        if (!editLog.isOpenForWrite()) {
            // 对 editlog 进行恢复操作，与之前的 Active 节点同步 editlog 内容
            editLog.initJournalsForWrite();
            editLog.recoverUnclosedStreams();
        }
    }
}
```



```

editLogTailer.catchupDuringFailover();

// 由于已经切换为 Active 状态的 Namenode, 所以不再延迟处理数据块
blockManager.setPostponeBlocksFromFuture(false);
// 新启动的 Active Namenode 需要等待所有 Datanode 更新心跳后
// 再触发删除与复制操作
blockManager.getDatanodeManager().markAllDatanodesStale();
// 清除 blockManager 中的所有队列
blockManager.clearQueues();
// 处理之前由于 Namespace 信息不完整而延迟处理的数据块
blockManager.processAllPendingDNMessages();

// 重新生成复制队列以及删除队列
if (!isInSafeMode()) {
    initializeReplQueues();
}

// 读取最新的 txid, 重置 editlog 的 txid, 之后打开 editlog
long nextTxId = getFSImage().getLastAppliedTxId() + 1;
editLog.setNextTxId(nextTxId);
getFSImage().editLog.openForWrite();
}

dir.enableQuotaChecks();
// 更新所有租约, 并启动租约检查线程
if (haEnabled) {
    leaseManager.renewAllLeases();
}
leaseManager.startMonitor();
startSecretManagerIfNecessary();

// 启动 Namenode 资源监控线程
this.nnrmthread = new Daemon(new NameNodeResourceMonitor());
nnrmthread.start();

// 启动 nnEditLogRoller 线程
nnEditLogRoller = new Daemon(new NameNodeEditLogRoller(
    editLogRollerThreshold, editLogRollerInterval));
nnEditLogRoller.start();

if (lazyPersistFileScrubIntervalSec > 0) {
    lazyPersistFileScrubber = new Daemon(new LazyPersistFileScrubber(
        lazyPersistFileScrubIntervalSec));
    lazyPersistFileScrubber.start();
}

// 启动 cacheManager 的监控线程
cacheManager.startMonitorThread();
blockManager.getDatanodeManager().setShouldSendCachingCommands(true);

```

```
    } finally {  
        startingActiveService = false;  
        checkSafeMode();  
        writeUnlock();  
    }  
}
```

transitionToStandby()

transitionToStandby()方法用于将 Active 状态的 Namenode 转换为 Standby 状态。这个方法的实现逻辑与 transitionToActive()是完全一致的，请读者参考上一节的分析，这里就不再赘述了。

getServiceStatus()

getServiceStatus()方法用于获取当前 Namenode 服务的状态，包括当前节点是处于 Active 还是 Standby 状态，以及当前节点能否切换为 Active 状态，不能切换为 Active 状态的原因等信息，这些信息都通过 HATServiceStatus 类封装。我们看一下 Namenode.getServiceStatus()方法的具体实现。

```
synchronized HATServiceStatus getServiceStatus()  
    throws ServiceException, AccessControlException {  
    namesystem.checkSuperuserPrivilege();  
    if (!haEnabled) {  
        throw new ServiceException("HA for namenode is not enabled");  
    }  
    if (state == null) {  
        return new HATServiceStatus(HATServiceState.INITIALIZING);  
    }  
    HATServiceState retState = state.getServiceState();  
    HATServiceStatus ret = new HATServiceStatus(retState);  
    if (retState == HATServiceState.STANDBY) {  
        String safemodeTip = namesystem.getSafeModeTip();  
        // Standby 状态，但是处于 safeMode 状态，则不可以切换为 Active 状态  
        if (!safemodeTip.isEmpty()) {  
            ret.setNotReadyToBecomeActive(  
                "The NameNode is in safemode. " +  
                safemodeTip);  
        } else { // 否则可以随时切换为 Active 状态  
            ret.setReadyToBecomeActive();  
        }  
    } else if (retState == HATServiceState.ACTIVE) {  
        // 对于 Active 状态的节点，随时可以切换为 Active 状态  
        ret.setReadyToBecomeActive();  
    } else {  
        ret.setNotReadyToBecomeActive("State is " + state);  
    }  
    return ret;  
}
```

(2) HAAdmin

HA 管理命令 (请参考管理命令小节) 的执行是由 HAAdmin 类负责的, HAAdmin 的 runCmd() 方法会对管理命令进行判断, 如果是 “-transitionToActive” 参数, 则调用 transitionToActive() 方法; 如果是 “-checkHealth” 参数, 则调用 checkHealth() 方法; 同理, 对于其他所有参数都会调用同名的私有方法。

这里我们重点关注 failover() 方法, 对应于管理命令参数 “-failover”, 也就是进行故障转移和切换 Namenode 状态的方法。failover() 方法首先会解析命令行参数, 从命令行中提取出执行切换操作的源节点与目标节点, 并将这两个节点封装成两个 HATarget 对象。由于 DFSHAAdmin 命令是在客户端发起的, 所以 HATarget 实际上包装了发生主从切换的两个 Namenode 的 HATargetProtocol 的代理。然后 failover() 方法会构造 FailoverController 对象用于控制主从切换, 最后调用 FailoverController.failover() 执行切换操作。

```
private int failover(CommandLine cmd)
    throws IOException, ServiceFailedException {
    // 解析命令行是否开启了 forceFence 以及 forceActive
    boolean forceFence = cmd.hasOption(FORCEFENCE);
    boolean forceActive = cmd.hasOption(FORCEACTIVE);
    // 解析源节点以及目标节点
    HATarget fromNode = resolveTarget(args[0]);
    HATarget toNode = resolveTarget(args[1]);

    // 如果是自动切换模式, 则不可以设置 forceFence 以及 forceActive
    if (fromNode.isAutoFailoverEnabled()) {
        if (forceFence || forceActive) {
            return -1;
        }
        return gracefulFailoverThroughZKFCs(toNode);
    }

    // 构造 FailoverController 对象, 并调用 failover() 方法执行切换操作
    FailoverController fc = new FailoverController(getConf(),
        requestSource);
    try {
        fc.failover(fromNode, toNode, forceFence, forceActive);
        out.println("Failover from "+args[0]+" to "+args[1]+" successful");
    } catch (FailoverFailedException ffe) {
        errOut.println("Failover failed: " + ffe.getLocalizedMessage());
        return -1;
    }
    return 0;
}
```

failover() 方法会调用 FailoverController.failover() 方法执行切换操作。FailoverController.failover() 方法会调用 HATargetProtocol 的 transitionToStandby() 以及 transitionToActive() 方法将源节点切换为 Standby 状态, 将目标节点切换为 Active 状态, 同时在将目标节点切换至 Active 状态前

Hadoop 2.X HDFS 源码剖析

执行 fencing 操作。如果目标节点切换 Active 状态失败，则执行回滚操作。HAResourceProtocol 的调用会由对应的 Namenode 响应，这部分代码我们在 HAResourceProtocol 小节中已经介绍了，读者可以参考。FailoverController.failover()方法的代码如下：

```
public void failover(HAResourceTarget fromSvc,
                    HAResourceTarget toSvc,
                    boolean forceFence,
                    boolean forceActive)
    throws FailoverFailedException {
    // 调用 preFailoverChecks() 进行切换前的检查，例如源节点和目标节点是否为同一个节
    // 点，目标节点是否已经处于 Active 状态，目标节点能否被设置为 Active 状态，目标
    // 节点是否有足够的磁盘资源切换为 Active 状态
    preFailoverChecks(fromSvc, toSvc, forceActive);

    // tryGracefulFence() 方法调用 RPC 接口 HAResourceProtocol.transitionToStandby()
    // 将源节点切换为 Standby 状态
    boolean tryFence = true;
    if (tryGracefulFence(fromSvc)) {
        tryFence = forceFence;
    }

    // 如果需要强制执行 fencing 操作则执行，如果 fencing 操作出现错误，则抛出异常
    if (tryFence) {
        if (!fromSvc.getFencer().fence(fromSvc)) {
            throw new FailoverFailedException("Unable to fence " +
                fromSvc + ". Fencing failed.");
        }
    }

    // 调用 HAResourceProtocol.transitionToActive() 将目标节点切换到 Active 状态
    boolean failed = false;
    Throwable cause = null;
    try {
        HAResourceProtocolHelper.transitionToActive(
            toSvc.getProxy(conf, rpcTimeoutToNewActive),
            createReqInfo());
    } catch (ServiceFailedException sfe) {
        LOG.error("Unable to make " + toSvc + " active (" +
            sfe.getMessage() + "). Failing back.");
        failed = true;
        cause = sfe;
    } catch (IOException ioe) {
        failed = true;
        cause = ioe;
    }

    // 将目标节点切换为 Active 状态失败，则考虑回滚
    // 如果没有 fencing 源节点，则回滚到原来的状态
```

```

// 如果源节点已经被强制 fencing, 则抛出异常, 因为源节点进程已经被终止
// 已经无法回滚到初始状态
if (failed) {
    String msg = "Unable to failover to " + toSvc;
    if (!tryFence) {
        try {
            // 将 toSvc 与 fromSvc 调换, 调用 failover() 方法回滚到原先的状态
            failover(toSvc, fromSvc, true, true);
        } catch (FailoverFailedException ffe) {
            msg += ". Failback to " + fromSvc +
                " failed (" + ffe.getMessage() + ")";
            LOG.fatal(msg);
        }
    }
    throw new FailoverFailedException(msg, cause);
}
}

```

至此, 用户通过管理员命令 DFSHAAdmin 执行 failover 切换的流程就介绍完了。对于其他操作的实现, 读者可以参考 HAAdmin 类中的方法, 都比较简单, 这里就不再赘述了。

(3) Quorum Journal

HDFS-1263 和相关的 JIRA 添加了 HDFS Namenode 高可用性支持, 但是所有的 HA 实现方案都依赖于一个保存 editlog 的共享存储。这个共享存储必须是高可用的, 并且能够被集群中的所有 Namenode 同时访问。

在 Quorum Journal 模式之前, HDFS 中使用最多的共享存储方案是 NAS+NFS。但是这种方案有个缺点, 就是为了预防脑裂的情况, 它要求有一个互斥脚本在 Namenode 发生故障切换时关闭上一个活动节点, 或者阻止上一个活动节点访问共享存储。为了解决这个问题, cloudera 提供了 Quorum Journal 设计方案, 这是一个基于 Paxos 算法实现的 HA 方案, 图 3-66 给出了 Quorum Journal 结构图。

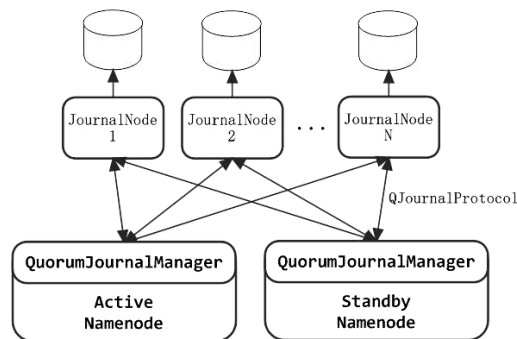


图 3-66 Quorum Journal 结构图

如图 3-66 所示, Quorum Journal 方案中有两个重要的组件。

- **JournalNode (JN)**: 运行在 N 台独立的物理机器上, 它将 `editlog` 文件保存在 `JournalNode` 的本地磁盘上, 同时 `JournalNode` 还对外提供 RPC 接口 `QJournalProtocol` 以执行远程读写 `editlog` 文件的功能。
- **QuorumJournalManager (QJM)**: 运行在 `Namenode` 上 (目前 HA 集群中只有两个 `Namenode`, `Active & Standby`), 通过调用 RPC 接口 `QJournalProtocol` 中的方法向 `JournalNode` 发送写入、互斥、同步 `editlog`。

`Quorum Journal` 方案依赖于这样一个概念: HDFS 集群中有 $2N+1$ 个 JN 存储 `editlog` 文件, 这些 `editlog` 文件是保存在 JN 的本地磁盘上的。每个 JN 对 QJM 暴露 RPC 接口 `QJournalProtocol`, 允许 `Namenode` 读写 `editlog` 文件。当 `Namenode` 向共享存储写入 `editlog` 文件时, 它会通过 QJM 向集群中的所有 JN 发送写 `editlog` 文件请求, 当有一半以上的 ($\geq N+1$) JN 返回写操作成功时即认为该次写成功。这个原理是基于 Paxos 算法的, 集群能容忍最多有 N 台机器挂掉, 如果多于 N 台挂掉, 这个算法就失效了。

使用 `Quorum Journal` 实现的 HA 方案有如下好处。

- JN 进程可以运行在普通的 PC 上, 而无须配置专业的共享存储硬件。
- 不需要实现单独的 fencing 机制, `Quorum Journal` 模式中内置了 fencing 功能。
- `Quorum Journal` 不存在单点故障, 集群中有 $2N+1$ 个 `JournalNode`, 可以允许有 N 个 `JournalNode` 死亡。
- JN 不会因为其中一台机器的延迟而影响整体的延迟, 而且也不会因为 JN 数量的增多而影响性能 (因为 `Namenode` 向 `JournalNode` 发送日志是并行的)。

本节我们就介绍基于 `Quorum Journal` 的 HA 方案。

QuorumJournalManager 实现

当 `Namenode` 启动后, 并且转换为 `Active` 状态时, 会调用 `FSEditLog.initJournalsForWrite()` 方法初始化存放 `editlog` 文件的存储, 这部分内容我们在 `FSEditLog` 类小节已经介绍过了。对于基于 QJM 的共享存储方式, `FSEditLog` 会创建一个 `QuorumJournalManager` 对象管理基于 `Quorum Journal` 的共享存储, 然后将这个对象加入 `FSEditLog.journalSet` 集合统一管理。本节我们就学习 `QuorumJournalManager` 的代码实现。

首先看一下 `QuorumJournalManager` 的构造方法, 它初始化了 QJM 的 `uri`、集群的 `nsinfo` 等一系列变量, 然后构造了一个 `AsyncLoggerSet` 对象管理当前 `QuorumJournalManager` 与集群中所有 `JournalNode` 之间的通信。

```
QuorumJournalManager(Configuration conf,
    URI uri, NamespaceInfo nsInfo,
    AsyncLogger.Factory loggerFactory) throws IOException {
    Preconditions.checkArgument(conf != null, "must be configured");

    this.conf = conf;
    this.uri = uri;
    this.nsInfo = nsInfo;
```

```

// 构造 AsyncLoggerSet 对象，维护与集群中所有 JournalNode 之间的连接
this.loggers = new AsyncLoggerSet(createLoggers(loggerFactory));
this.connectionFactory = URLConnectionFactory
    .newDefaultURLConnectionFactory(conf);

// 配置各种超时时间
// ...
}

```

在介绍 AsyncLoggerSet 对象之前，我们先看一下 AsyncLogger 类的定义。AsyncLogger 是一个接口类，它定义了 QuorumJournalManager 与集群中一个 JournalNode 之间的所有异步通信接口。AsyncLogger 使用适配器模式，它包装了 QJournalProtocol 接口，并将 QJournalProtocol 接口适配成带有异步调用机制的接口。

AsyncLogger 接口的实现类是 IPCLoggerChannel 类，IPCLoggerChannel 类定义了一个线程池对象 singleThreadExecutor，IPCLoggerChannel 类会在 singleThreadExecutor 对象上调用 JournalNode 对应的 QJournalProtocol 代理类的接口方法，并将返回结果使用 ListenableFuture 封装，也就实现了异步 RPC 调用的功能。这里我们以 IPCLoggerChannel.startLogSegment() 方法为例，IPCLoggerChannel.startLogSegment() 方法对 QJournalProtocol.startLogSegment() 方法进行了封装，它在 IPCLoggerChannel 的线程池对象 singleThreadExecutor 上提交了一个 Callable 任务，这个 Callable 任务在 JN 对应的 QJournalProtocol 代理对象上调用了 startLogSegment() 请求，并将线程执行结果封装在一个 ListenableFuture 对象中。这里还要注意，AsyncLogger 类定义了 outOfSync 字段，outOfSync 用于标识当前 JN 上写 editlog 的操作是否出现了错误，例如在 editlog 段落（segment）中丢失了部分数据等。一旦出现这种情况（outOfSync==true），AsyncLogger 对象就不可以继续当前 editlog 段落中写入新的数据了，直到 Namenode 调用 startLogSegment() 方法重新开启了一个新的 editlog 段落，startLogSegment() 方法会重置 outOfSync 字段。startLogSegment() 方法的代码如下：

```

public ListenableFuture<Void> startLogSegment(final long txid,
    final int layoutVersion) {
    return singleThreadExecutor.submit(new Callable<Void>() {
        @Override
        public Void call() throws IOException {
            // 在 JN 对应的 QJournalProtocol 代理对象上调用了 startLogSegment() 请求
            getProxy().startLogSegment(createReqInfo(), txid, layoutVersion);
            synchronized (IPCLoggerChannel.this) {
                if (outOfSync) { // 重置 outOfSync 标志位
                    outOfSync = false;
                }
            }
            return null;
        }
    });
}

```

了解了 AsyncLogger 类，我们再来看 AsyncLoggerSet。AsyncLoggerSet 类其实就是一组 AsyncLogger 对象的集合。图 3-67 给出了 AsyncLoggerSet、AsyncLogger 以及 QJournalProtocol 之间的关系。

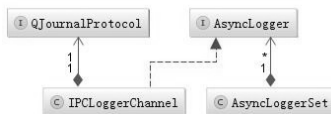


图 3-67 AsyncLoggerSet、AsyncLogger 以及 QJournalProtocol 之间的关系

AsyncLoggerSet 对集合中所有 AsyncLogger 对象上的调用进行了封装，并使用一个 QuorumCall 对象保存所有 JournalNode 返回的结果，也就是实现了 QuorumJournalManager 与集群中所有 JournalNode 的异步通信。这里我们还是以 AsyncLoggerSet.startLogSegment() 方法为例，AsyncLoggerSet.startLogSegment() 方法会遍历 AsyncLoggerSet 中保存的所有 AsyncLogger 对象，并在这些对象上调用 AsyncLogger.startLogSegment() 方法。我们知道 AsyncLogger 接口的返回结果都是 ListenableFuture 类型的，AsyncLoggerSet.startLogSegment() 方法会构造一个 Map 对象保存 AsyncLogger 对象，以及在这个 AsyncLogger 对象上调用 startLogSegment() 方法返回的 ListenableFuture 对象。之后 startLogSegment() 方法会构造一个 QuorumCall 对象封装这个保存了调用结果的 Map 对象并返回。AsyncLoggerSet.startLogSegment() 方法的代码如下：

```

public QuorumCall<AsyncLogger, Void> startLogSegment(
    long txid, int layoutVersion) {
    Map<AsyncLogger, ListenableFuture<Void>> calls = Maps.newHashMap();
    for (AsyncLogger logger : loggers) { // 遍历所有的 AsyncLogger 对象并调用
startLogSegment() 方法
        calls.put(logger, logger.startLogSegment(txid, layoutVersion));
    }
    return QuorumCall.create(calls);
}

```

QuorumCall 这个类包装了 AsyncLoggerSet 的整个异步调用过程。由上面的分析我们知道，每次 AsyncLoggerSet 对象向 $2N+1$ 个 JN 发送写日志请求都是异步的（由 AsyncLogger 对象实现），AsyncLoggerSet 在请求发出之后并不同步等待每个 JN 的返回值，而是在每个 AsyncLogger 返回的 ListenableFuture 对象上注册回调函数，每当调用返回时，都会执行回调函数把 QuorumCall 的响应计数加 1（如果返回是 success，则把 success 计数加 1；如果返回是 failure，则把 failure 计数加 1）。这部分代码是在 QuorumCall.create() 方法中实现的。

```

static <KEY, RESULT> QuorumCall<KEY, RESULT> create(
    Map<KEY, ? extends ListenableFuture<RESULT>> calls) {
    final QuorumCall<KEY, RESULT> qr = new QuorumCall<KEY, RESULT>();
    // 遍历所有的 ListenableFuture 对象
    for (final Entry<KEY, ? extends ListenableFuture<RESULT>> e : calls.entrySet()) {
        // 在 ListenableFuture 对象上添加回调函数
        Futures.addCallback(e.getValue(), new FutureCallback<RESULT>() {

```



```

@Override
public void onFailure(Throwable t) {
    qr.addException(e.getKey(), t); // 如果异常，则异常计数加1
}

@Override
public void onSuccess(RESULT res) {
    qr.addResult(e.getKey(), res); // 如果成功返回，则成功计数加1
}
});
}
return qr;
}

```

有了回调函数，QuorumJournalManager 端只需要发出请求，然后循环检测 QuorumCall 对象是否有足够的 success 响应、足够的 exception 响应或者 timeout 即可。这部分代码是在 QuorumCall.waitFor()方法中执行的。

```

public synchronized void waitFor(
    int minResponses, int minSuccesses, int maxExceptions,
    int millis, String operationName)
    throws InterruptedException, TimeoutException {
    long st = Time.monotonicNow();
    long nextLogTime = st + (long)(millis * WAIT_PROGRESS_INFO_THRESHOLD);
    long et = st + millis;
    while (true) {
        checkAssertionErrors();
        // 有足够的响应，则返回
        if (minResponses > 0 && countResponses() >= minResponses) return;
        // 有足够的成功响应，则返回
        if (minSuccesses > 0 && countSuccesses() >= minSuccesses) return;
        // 有足够的异常响应，则返回
        if (maxExceptions >= 0 && countExceptions() > maxExceptions) return;
        long now = Time.monotonicNow();

        // 计算等待时间
        if (now > nextLogTime) {
            long waited = now - st;
            nextLogTime = now + WAIT_PROGRESS_INTERVAL_MILLIS;
        }
        long rem = et - now;
        if (rem <= 0) {
            throw new TimeoutException();
        }
        rem = Math.min(rem, nextLogTime - now);
        rem = Math.max(rem, 1);
        wait(rem); // 等待响应
    }
}

```

了解了 AsyncLoggerSet 的代码实现，再来看 QuorumJournalManager 就比较简单了，QuorumJournalManager 实现了 JournalManager 接口中定义的管理 editlog 存储的所有方法。

当 Namenode 调用 QuorumJournalManager 中的 JournalManager 接口方法时，QuorumJournalManager 会在 AsyncLoggerSet 上调用对应的方法将请求发送到集群中的所有 JN 上，然后等待集群中多数 JN 响应后，将执行情况返回给 Namenode。这里我们还是以 startLogSegment() 方法为例，当 Namenode 调用 QuorumJournalManager.startLogSegment() 在共享存储上开启一个新的 editlog 段落时，QuorumJournalManager.startLogSegment() 会调用 AsyncLoggerSet.startLogSegment() 将这个请求前转到集群中的所有 JN 上。AsyncLoggerSet.startLogSegment() 方法会返回一个 QuorumCall 对象，之后 startLogSegment() 方法会调用 AsyncLoggerSet.waitForWriteQuorum() 方法等待所有异步请求的返回值，waitForWriteQuorum() 方法调用了 QuorumCall.waitFor() 方法统计 JN 的返回情况，如果大多数 JN 返回成功，则 waitForWriteQuorum() 方法返回成功；如果执行成功的 JN 数量不足，则抛出异常。最后 startLogSegment() 方法会构造 QuorumOutputStream 对象并返回。QuorumJournalManager.startLogSegment() 方法的代码如下所示，它调用的 AsyncLoggerSet.startLogSegment() 以及 QuorumCall.waitFor() 方法请参考前面的分析。

```
public EditLogOutputStream startLogSegment(long txId, int layoutVersion)
    throws IOException {
    Preconditions.checkState(isActiveWriter,
        "must recover segments before starting a new one");
    // 调用 AsyncLoggerSet.waitForWriteQuorum() 将请求发送到所有 JN 上
    QuorumCall<AsyncLogger, Void> q = loggers.startLogSegment(txId,
        layoutVersion);
    // 等待 JN 返回执行结果
    loggers.waitForWriteQuorum(q, startSegmentTimeoutMs,
        "startLogSegment(" + txId + ")");
    // 构造 QuorumOutputStream 对象，并返回
    return new QuorumOutputStream(loggers, txId,
        outputBufferCapacity, writeTxnsTimeoutMs);
}
```

图 3-68 给出了 QuorumJournalManager.startLogSegment() 方法流程图，读者可以结合上面的代码分析一起学习。

至此，QuorumJournalManager 的实现就介绍完了。至于 QuorumJournalManager 中定义的其他方法，我们在后面的写流程、读流程以及恢复流程中单独介绍。同样的，startLogSegment() 返回的 QuorumOutputStream 对象，我们在 QJM 的写流程中详细介绍。

互斥机制

当 HA 集群发生 Namenode 异常切换时，需要在共享存储上 fencing 上一个活动节点以保证该节点不能再向共享存储写入 editlog。基于 Quorum Journal 模式的 HA 提供了 epoch number 来解决互斥 (fencing) 问题，这个概念在很多分布式文献中都能找到（例如 Paxos、ZAB 等）。epoch number 具有如下一些性质。

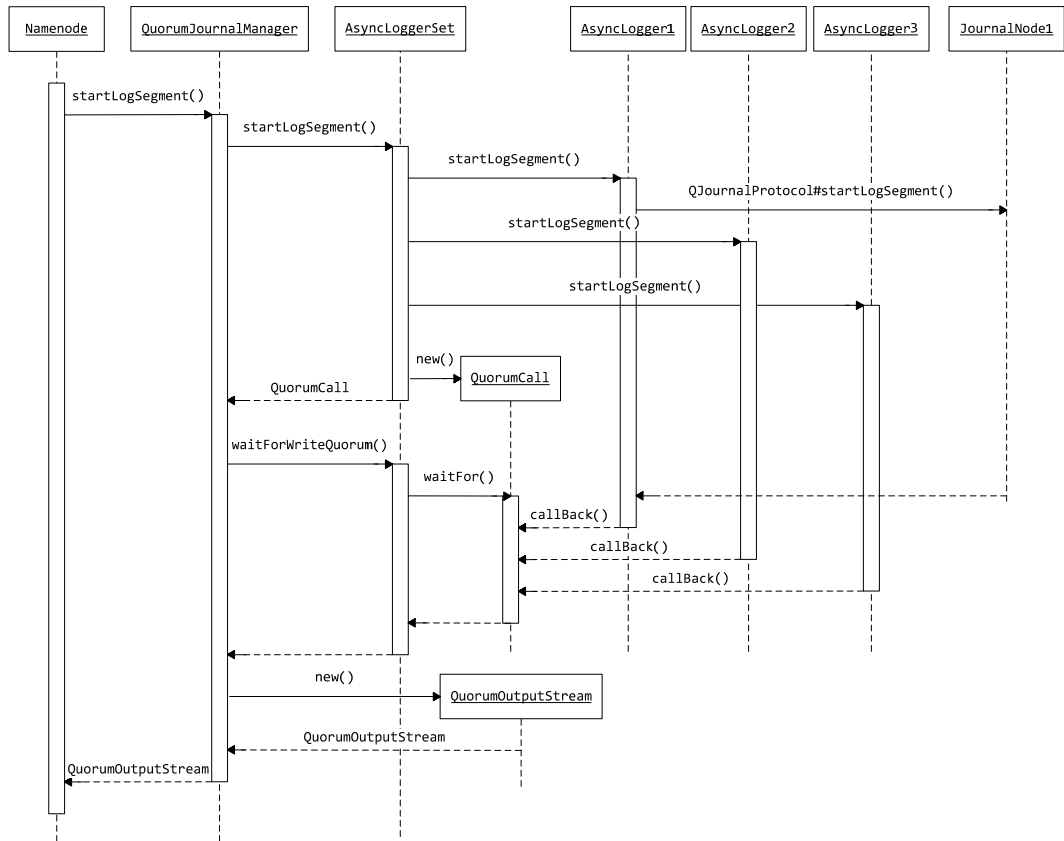


图 3-68 QuorumJournalManager.startLogSegment()方法流程图

- 当一个 Namenode 变为活动状态时，会分配给它一个 epoch number。
- 每个 epoch number 都是唯一的，没有任意两个 Namenode 有相同的 epoch number。
- epoch number 定义了 Namenode 写 editlog 文件的顺序。对于任意两个 Namenode，拥有更大 epoch number 的 Namenode 被认为是活动节点

当一个 Namenode 切换为活动状态时，它的 QJM 会向所有 JN 发送 QJournalProtocol.getJournalState()请求以获取该 JN 的 lastPromisedEpoch 变量值，lastPromisedEpoch 变量保存了该 JN 认为的集群中活动的 Namenode 对应的 epoch number 值。当 QJM 接收到集群中多于一半的 JN 回复后，它会将接收到的最大值加 1 并保存到 myEpoch 变量中，之后 QJM 会调用 QJournalProtocol.newEpoch(myEpoch)方法向所有 JN 发起更新 epoch number 请求。每个 JN 都会比较 QJM 提交的 myEpoch 变量，以及当前 JN 保存的 lastPromisedEpoch 变量，如果新的 myEpoch 较大，则更新 JN 的 lastPromisedEpoch 为新值，并且返回更新成功；如果小，则返回更新失败。如果 QJM 接收到超过一半的 JN 返回成功，则设置它的 epoch number 为 myEpoch；

否则，它中止尝试成为一个活动的 Namenode，并抛出 `IOException` 异常。

```
Map<AsyncLogger, NewEpochResponseProto> createNewUniqueEpoch()
    throws IOException {
    Preconditions.checkState(!loggers.isEpochEstablished(),
        "epoch already created");

    // 获取集群中所有 JN 的 lastPromisedEpoch 变量
    Map<AsyncLogger, GetJournalStateResponseProto> lastPromises =
        loggers.waitForWriteQuorum(loggers.getJournalState(),
            getJournalStateTimeoutMs, "getJournalState()");

    long maxPromised = Long.MIN_VALUE;
    for (GetJournalStateResponseProto resp : lastPromises.values()) {
        maxPromised = Math.max(maxPromised, resp.getLastPromisedEpoch());
    }
    assert maxPromised >= 0;
    // 统计出最大的 epoch number 加 1 之后作为这个 QJM 的 epoch number
    long myEpoch = maxPromised + 1;
    // 调用 newEpoch() 更新 JN 上的 epoch number 为 myEpoch
    Map<AsyncLogger, NewEpochResponseProto> resps =
        loggers.waitForWriteQuorum(loggers.newEpoch(nsInfo, myEpoch),
            newEpochTimeoutMs, "newEpoch(" + myEpoch + ")");

    loggers.setEpoch(myEpoch);
    return resps;
}
```

当活动的 Namenode 成功获取并更新了 epoch number 后，调用任何修改 editlog 的 RPC 请求都必须携带 epoch number。当 RPC 请求到达 JN 后（除了 `newEpoch()` 请求），JN 会将请求者的 epoch number 与自己保存的 `lastPromisedEpoch` 变量做比较，如果请求者的 epoch number 更大，JN 就会更新自己的 `lastPromisedEpoch` 变量，并执行对应的操作；如果请求者的 epoch number 更小，JN 就会拒绝这次请求。当集群中的大多数 JN 拒绝了请求时，这次操作就失败了。考虑如下情况，当 HDFS 集群发生 Namenode 错误切换后，原来 Standby Namenode 会将集群的 epoch number 加 1 之后更新。这样原来的 Active Namenode 的 epoch number 肯定小于这个值，当这个节点执行写 editlog 操作时，由于 JN 节点不接收 epoch number 小于 `lastPromisedEpoch` 的写请求，所以这次写请求会失败，也就达到了 fencing 的目的。

写流程

Active Namenode 会将 editlog 写入集群中的 JN 上，如图 3-69 所示，QJM 会按照以下流程执行写操作。

- 将 editlog 输出流中缓存的数据写入 JN，对于集群中的每一个 JN 都存在一个独立的线程调用 RPC 接口 `QJournalProtocol.journal()` 向 JN 写入数据。
- 当 JN 收到 `QJournalProtocol.journal()` 请求后，JN 会执行如下操作：①验证 epoch number 是否正确；②确认写入数据对应的 txid 是否连续；③将数据持久化到 JN 的

本地磁盘；④向 QJM 发送正确的响应。

- QJM 等待集群 JN 的响应，如果多数 JN 返回成功，则写操作成功；否则写操作失败，QJM 会抛出异常。

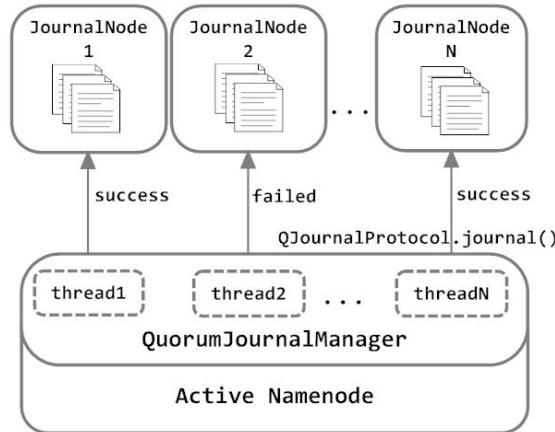


图 3-69 写 editlog 流程图

在 FSEditLog 类小节中我们介绍了，Namenode 会调用 FSEditLog.startLogSegment()方法初始化 editlog 文件的输出流，然后使用输出流对象向 editlog 文件写入数据。对于 Quorum Journal 的共享存储，FSEditLog.startLogSegment()会调用 QuorumJournalManager.startLogSegment()在共享存储上打开一个 editlog 段落文件，创建 QuorumOutputStream 输出流对象并返回。这部分内容我们在 QuorumJournalManager 实现小节中已经介绍了。

获取了 QuorumOutputStream 输出流对象后，Namenode 会调用 QuorumOutputStream.write()方法向 editlog 文件中写入数据，QuorumOutputStream 的实现类似于 EditLogFileOutputStream (请参考 EditLogFileOutputStream 小节的说明)，底层也使用了 EditsDoubleBuffer 双缓冲区。数据会先写入其中一个缓冲区中，然后在调用 QuorumOutputStream.flushAndSync()时将缓冲区中的数据发送给 JN。flushAndSync()方法调用了 AsyncLoggerSet.sendEdits()方法将缓冲区中的数据发送给集群中的所有 JN，然后调用 AsyncLoggerSet.waitForWriteQuorum()方法等待集群中的大多数 JN 响应写请求。完成了写操作后，flushAndSync()方法会调用 AsyncLoggerSet.setCommittedTxId()记录最近提交的 txid。AsyncLoggerSet.flushAndSync()方法的代码如下：

```

protected void flushAndSync(boolean durable) throws IOException {
    int numReadyBytes = buf.countReadyBytes();
    if (numReadyBytes > 0) {
        int numReadyTxns = buf.countReadyTxns();
        long firstTxToFlush = buf.getFirstReadyTxId();
        assert numReadyTxns > 0;
        DataOutputBuffer bufToSend = new DataOutputBuffer(numReadyBytes);
        buf.flushTo(bufToSend);
        assert bufToSend.getLength() == numReadyBytes;
    }
}

```

Hadoop 2.X HDFS 源码剖析

```
byte[] data = bufToSend.getData();
assert data.length == bufToSend.getLength();
// 调用 sendEdits() 方法向集群中的所有 JN 写入数据
QuorumCall<AsyncLogger, Void> qcall = loggers.sendEdits(
    segmentTxId, firstTxToFlush,
    numReadyTxns, data);
loggers.waitForWriteQuorum(qcall, writeTimeoutMs, "sendEdits");

loggers.setCommittedTxId(firstTxToFlush + numReadyTxns - 1);
}
}
```

flushAndSync()调用 AsyncLoggerSet.sendEdits()方法向集群中的所有 JN 写入数据, 下面我们看一下 AsyncLoggerSet.sendEdits()方法的实现。AsyncLoggerSet.sendEdits()方法会遍历所有的 AsyncLogger 对象并调用 sendEdits() 方法, AsyncLogger.sendEdits() 方法会调用 QJournalProtocol.journal()方法向指定 JN 写入 editlog 文件, 如果 JN 在写入 editlog 时出现错误, 则 sendEdits()方法会将 AsyncLogger.outOfSync 字段设置为 true, 之后 AsyncLogger 都不可以再向这个 editlog 段落文件写入数据了, 直到调用 AsyncLogger.startLogSegment()方法开启了新的 editlog 段落时才可以恢复写操作。

```
public ListenableFuture<Void> sendEdits(
    final long segmentTxId, final long firstTxnId,
    final int numTxns, final byte[] data) {
    // ...
    ListenableFuture<Void> ret = null;
    try {
        ret = singleThreadExecutor.submit(new Callable<Void>() {
            @Override
            public Void call() throws IOException {
                throwIfOutOfSync();
                long rpcSendTimeNanos = System.nanoTime();
                try {
                    // 调用 journal() 方法向 JN 写入数据
                    getProxy().journal(createReqInfo(),
                        segmentTxId, firstTxnId, numTxns, data);
                } catch (IOException e) {
                    synchronized (IPCLoggerChannel.this) {
                        // 如果出现错误, 则将 outOfSync 设置为 true
                        // AsyncLogger 不可以继续向这个 editlog 段落写入数据了
                        outOfSync = true;
                    }
                    throw e;
                } finally {
                    // ...
                }
                // ...
                return null;
            }
        });
    }
```

```

    });
} finally {
    // ...
}
return ret;
}

```

至此，Namenode 向 JN 写入 editlog 的流程就结束了。

读流程

Standby Namenode 会从 JN 读取 editlog，然后与 Standby Namenode 的命名空间合并以保持和 Active Namenode 命名空间的同步。如图 3-70 所示，当 Standby Namenode 从 JN 读取 editlog 时，它会首先发送 RPC 请求 `QJournalProtocol.getEditLogManifest()` 到集群中的所有 JN 上。JN 接收到这个请求后会将 JN 本地存储上保存的所有 FINALIZED 状态的 editlog 段落文件信息返回，之后 QJM 会为所有 JN 返回的 editlog 段落文件构造输入流对象，并将这些输入流对象合并到一个 `RedundantEditLogInputStream` 对象中，这样 Standby Namenode 就可以使用这个对象从任意一个 JN 上读取每个 editlog 段落了。如果其中一个 JN 失败了，`RedundantEditLogInputStream` 会自动切换到另一个保存了这个 editlog 段落的 JN 上。

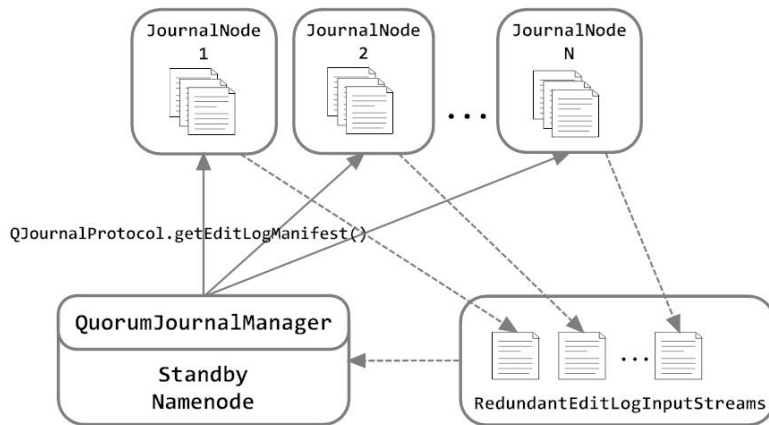


图 3-70 读 editlog 流程图

FSEditLog 会调用 `QuorumJournalManager.selectInputStreams()` 方法打开共享存储上的 editlog 文件，`selectInputStreams()` 方法首先会发送 RPC 请求 `QJournalProtocol.getEditLogManifest()` 到集群中的所有 JN 上，以获取集群中所有 JN 上保存的 editlog 段落文件信息。之后对于 JN 上保存的每一个 editlog 文件，`selectInputStreams()` 都会构造一个 `EditLogFileInputStream` 对象，然后将这些对象都封装在一个 `RedundantEditLogInputStream` 输入流对象中。

```

public void selectInputStreams(Collection<EditLogInputStream> streams,
    long fromTxnId, boolean inProgressOk) throws IOException {
    // 调用 AsyncLoggerSet.getEditLogManifest() 向 JN 发送 getEditLogManifest() 请求
    QuorumCall<AsyncLogger, RemoteEditLogManifest> q =

```

```
        loggers.getEditLogManifest(fromTxnId, inProgressOk);
// 等待所有 JN 响应
Map<AsyncLogger, RemoteEditLogManifest> resps =
    loggers.waitForWriteQuorum(q, selectInputStreamsTimeoutMs,
        "selectInputStreams");
final PriorityQueue<EditLogInputStream> allStreams =
    new PriorityQueue<EditLogInputStream>(64,
        JournalSet.EDIT_LOG_INPUT_STREAM_COMPARATOR);
for (Map.Entry<AsyncLogger, RemoteEditLogManifest> e : resps.entrySet()) {
    AsyncLogger logger = e.getKey();
    RemoteEditLogManifest manifest = e.getValue();

    for (RemoteEditLog remoteLog : manifest.getLogs()) {
        URL url = logger.buildURLToFetchLogs(remoteLog.getStartTxId());
        // 为 JN 上保存的每一个 editlog 段落文件构造 EditLogFileInputStream 对象
        EditLogInputStream elis = EditLogFileInputStream.fromUrl(
            connectionFactory, url, remoteLog.getStartTxId(),
            remoteLog.getEndTxId(), remoteLog.isInProgress());
        // 将新构造的对象加入 allStreams 集合中暂存
        allStreams.add(elis);
    }
}

// 将所有 EditLogFileInputStream 封装在一个 RedundantEditLogInputStream
JournalSet.chainAndMakeRedundantStreams(streams, allStreams, fromTxnId);
}
```

获取了 `RedundantEditLogInputStream` 对象后, `Namenode` 就可以使用这个输入流执行读取操作了。

恢复流程

当 `Namenode` 发生主从切换时, 原来的 `Standby Namenode` 会接管共享存储并执行写 `editlog` 的操作。 `Standby Namenode` 在切换为 `Active Namenode` 前, 对于共享存储会执行如下操作。

- `fencing` 原来的 `Active Namenode`。这部分内容我们在互斥机制小节中已经介绍了。
- 恢复正在处理 (in-progress) 的 `editlog`。由于 `Namenode` 发生了主从切换, 集群中 `JN` 上的正在执行写入操作的 `editlog` 数据可能不一致。例如, 可能出现某些 `JN` 上的 `editlog` 正在写入, 但是当前 `Active Namenode` 发生了错误, 这时该 `JN` 上 `editlog` 文件的状态就与已经完成写入的 `JN` 不一致。在这种情况下, 需要对 `JN` 上所有状态不一致的 `editlog` 文件执行恢复操作, 将它们的数据同步一致, 并且将 `editlog` 文件转换为 `FINALIZED` 状态。
- `startLogSegment`, 由于不一致的 `editlog` 已经完成了恢复, 所以这时原来的 `Standby Namenode` 就可以切换为 `Active Namenode` 并执行写 `editlog` 的操作了, 也就是可以通过 `startLogSegment` 操作打开 `editlog` 文件, 然后执行写操作了。这部分代码是在 `QuorumJournalManager.startLogSegment()` 中实现的, 我们在 `QuorumJournalManager`

实现小节中已经介绍过了。

- 写 editlog。当 QJM 通过调用 startLogSegment()方法打开了共享存储上的 editlog 文件并获得了输出流对象后，就可以向 editlog 中写入数据了。这部分内容我们在写流程小节中已经介绍过了。
- finalizeLogSegment，当 Namenode 完成了对一个 editlog 日志段落的写操作后，会调用 QuorumJournalManager.finalizeLogSegment()提交这个日志段落。
- 重复至 startLogSegment 步骤。

通过 HATServiceProtocol 小节中的分析我们知道，当 Standby Namenode 切换为 Active Namenode 时会调用 FSNamesystem.startActiveService()方法启动 Active Namenode 的所有服务，startActiveService()会调用 QuorumJournalManager.recoverUnfinalizedSegments()来执行共享存储上的切换服务。recoverUnfinalizedSegments()方法执行了上面流程中的步骤一和步骤二，它调用了 createNewUniqueEpoch()方法执行 fencing 原来的 Active Namenode 操作。这部分内容我们在互斥机制小节中已经介绍过了。之后 recoverUnfinalizedSegments()方法会调用 recoverUnclosedSegment()恢复正在执行写操作的 editlog 文件 (in-progress)，也就是同步 JN 上 editlog 文件的状态。完成了 recoverUnfinalizedSegments()调用后，Namenode 就可以调用 QuorumJournalManager.startLogSegment() 开始 新 一个 editlog 段落的写操作了。recoverUnfinalizedSegments()方法的代码如下：

```
public void recoverUnfinalizedSegments() throws IOException {
    // fencing 原来的 Active Namenode
    Map<AsyncLogger, NewEpochResponseProto> resps = createNewUniqueEpoch();

    // createNewUniqueEpoch() 调用会返回 JN 上最新一个 editlog 段落的 txid
    long mostRecentSegmentTxId = Long.MIN_VALUE;
    for (NewEpochResponseProto r : resps.values()) {
        if (r.hasLastSegmentTxId()) {
            // 计算最近最新的 editlog 段落的 txid，以执行恢复操作
            mostRecentSegmentTxId = Math.max(mostRecentSegmentTxId,
                r.getLastSegmentTxId());
        }
    }

    // 调用 recoverUnclosedSegment() 方法执行恢复操作
    if (mostRecentSegmentTxId != Long.MIN_VALUE) {
        recoverUnclosedSegment(mostRecentSegmentTxId);
    }
    isActiveWriter = true;
}
```

createNewUniqueEpoch()方法我们在互斥机制小节中已经介绍过了，本节就重点介绍执行恢复操作的 recoverUnclosedSegment()方法。日志恢复操作可以分为如下几个阶段。

- 确定需要执行恢复操作的 editlog 段落：在执行恢复操作前，QJM 会执行 newEpoch()调用以产生新的 epoch number，JN 接收到这个请求后除了执行更新 epoch number 的

操作外，还会将该 JN 上保存的最新的 editlog 段落的 txid 返回。当集群中的大多数 JN 都发回了这个响应后，QJM 就可以确定出集群中最新的一个正在处理 editlog 段落的 txid，然后 QJM 就会对这个 txid 对应的 editlog 段落执行恢复操作了。

- 准备恢复：QJM 向集群中的所有 JN 发送 RPC 请求 `prepareRecovery()`，查询执行恢复操作的 editlog 段落文件在所有 JN 上的状态。这里的状态包括 editlog 文件是 in-progress 还是 FINALIZED 状态，以及 editlog 文件的长度。
- 接受恢复：QJM 接收到 JN 发回的响应后，会根据恢复算法选择执行恢复操作的源节点。然后 QJM 会发送 RPC 请求 `acceptRecovery()` 给每一个 JN，这个请求包含两部分信息：源 editlog 段落文件信息，以及供 JN 下载这个源 editlog 段落的 url。接收到这个 `acceptRecovery()` 请求的 JN 会执行如下操作：① 同步 editlog 段落文件，如果当前 JN 磁盘上的 editlog 段落文件与请求中段落文件的状态不同，则 JN 会从请求中的 url 上下载段落文件并替换磁盘上的 editlog 段落文件；② 持久化恢复元数据，JN 会将执行恢复操作的 editlog 段落文件的状态、触发恢复操作的 QJM 的 epoch number 等信息持久化到 JN 磁盘上。这样当再次有 `prepareRecovery()` 请求时，JN 会将这部分新的数据返回。当这些操作都成功执行后，JN 会返回成功响应给 QJM，如果集群中的大多数 JN 都返回了成功，则此次恢复操作执行成功。
- 完成（finalize）editlog 段落文件：到这步操作时，QJM 已经确定集群中大多数 JN 保存的 editlog 文件的状态已经一致了，并且 JN 持久化了恢复信息。这时 QJM 就可以调用 `finalizedLogSegment()` 请求，将这个 editlog 段落文件转换为 FINALIZED 状态。当 JN 接收到 `finalizedLogSegment()` 请求后，它会将 editlog 文件状态设置为 FINALIZED，同时删除持久化的恢复元数据，因为磁盘上保存的 editlog 文件信息已经是正确的了，不需要保存恢复的元数据了。

至此，一个完整的 editlog 段落恢复操作就完成了。上面描述的操作是在 `recoverUnclosedSegment()` 方法中执行的。`recoverUnclosedSegment()` 方法的代码如下：

```
private void recoverUnclosedSegment(long segmentTxId) throws IOException {
    // segmentTxId 在 recoverUnfinalizedSegments() 方法中已经计算出来了

    // 调用 AsyncLoggerSet.prepareRecovery() 向集群中的 JN 获取 editlog 段落文件状态
    QuorumCall<AsyncLogger, PrepareRecoveryResponseProto> prepare =
        loggers.prepareRecovery(segmentTxId);
    Map<AsyncLogger, PrepareRecoveryResponseProto> prepareResponses =
        loggers.waitForWriteQuorum(prepare, prepareRecoveryTimeoutMs,
            "prepareRecovery(" + segmentTxId + ")");

    // 使用 SegmentRecoveryComparator 比较所有 JN 返回的 editlog 段落文件的状态
    // 根据恢复算法选择一个最好的 editlog 文件，作为后续恢复操作的标准以及源文件
    Entry<AsyncLogger, PrepareRecoveryResponseProto> bestEntry = Collections.max(
        prepareResponses.entrySet(), SegmentRecoveryComparator.INSTANCE);
    AsyncLogger bestLogger = bestEntry.getKey();
    PrepareRecoveryResponseProto bestResponse = bestEntry.getValue();

    // ...
}
```

```

// 待恢复 editlog 段落信息
SegmentStateProto logToSync = bestResponse.getSegmentState();
assert segmentTxId == logToSync.getStartTxId();
// 构造筛选出的 editlog 文件的 url, 其他 JN 可以通过这个 url 读取源 editlog 文件
URL syncFromUrl = bestLogger.buildURLToFetchLogs(segmentTxId);

// 调用 AsyncLoggerSet.acceptRecovery() 向 JN 发送 acceptRecovery RPC 请求
QuorumCall<AsyncLogger,Void> accept=loggers.acceptRecovery(logToSync, syncFromUrl);
// 等待集群中大多数 JN 响应
loggers.waitForWriteQuorum(accept, acceptRecoveryTimeoutMs,
    "acceptRecovery(" + TextFormat.shortDebugString(logToSync) + ")");

// editlog 文件已经同步完成, 可以执行 finalized 操作了
QuorumCall<AsyncLogger, Void> finalize =
    loggers.finalizeLogSegment(logToSync.getStartTxId(), logToSync.getEndTxId());
loggers.waitForWriteQuorum(finalize, finalizeSegmentTimeoutMs,
    String.format("finalizeLogSegment(%s-%s)",
        logToSync.getStartTxId(),
        logToSync.getEndTxId()));
}

```

读者读到这里可能会有疑问：这个源 editlog 段落文件（对应的 JournalNode 作为源 JournalNode）是如何选择出来的呢？QJM 按照下面的算法选择源 editlog 段落文件，代码实现在 SegmentRecoveryComparator.compare()方法中。

- FINALIZED 状态的 editlog 段落文件优于 in-progress 状态的 editlog 段落文件。
- 如果存在多个 FINALIZED 状态的 editlog 段落文件，则这些文件的文件长度必须一致。
- 如果都处于 in-progress 状态，则选择 epoch number 大的文件。
- 如果都处于 in-progress 状态，且 epoch number 相同，则选择包含事务（transaction）较多的 editlog 文件。

（4）ZKFailoverController

在 HAAdmin 小节中我们介绍了用户如何手动执行 Failover 操作，Hadoop 在手动 Failover 的基础上还提供了自动 Failover 机制，自动 Failover 机制依赖于两个新增的网元：一个是 ZooKeeper 集群；一个是 ZKFailoverController（org.apache.hadoop.ha.ZKFailoverController）。由于篇幅的原因，这里我们就不再介绍 ZKFailoverController 的代码实现了，它涉及了与 ZooKeeper 集群的耦合，建议读者自己参考源码。

3.7.3 名字节点的启动

Namenode 实体在代码实现中主要对应于三个类，即 NameNode 类、NameNodeRpcServer 类以及 FSNamesystem 类。NameNodeRpcServer 类用于接收和处理所有的 RPC 请求，

FSNamesystem 类负责实现 Namenode 的所有逻辑，而 NameNode 类则负责管理 Namenode 配置、RPC 接口以及 HTTP 接口等。所以 Namenode 的启动操作是在 NameNode 类中执行的，具体的方法是 NameNode.main()方法，main()方法首先调用 createNameNode()创建一个 NameNode 对象，创建成功后调用 NameNode.join()方法等待 RPC 服务结束。main 方法的代码如下：

```
public static void main(String argv[]) throws Exception {
    if (DFSUtil.parseHelpArgument(argv, NameNode.USAGE, System.out, true)) {
        System.exit(0);
    }
    try {
        // 调用 createNameNode() 方法创建 NameNode 对象
        NameNode namenode = createNameNode(argv, null);
        if (namenode != null) {
            namenode.join();    // 等待 Namenode RPC 服务结束
        }
    } catch (Throwable e) {
        terminate(1, e);    // 出现异常则直接退出执行
    }
}

// join() 方法等待 RPC 服务执行完毕，在通常情况下这个服务是一直循环执行的
public void join() {
    try {
        rpcServer.join();
    } catch (InterruptedException ie) {
        LOG.info("Caught interrupted exception ", ie);
    }
}
```

createNameNode()方法会根据启动 Namenode 时传入的启动选项，调用对应的方法执行操作。

- **FORMAT**: 格式化当前 Namenode，调用 format()方法执行格式化操作。
- **FINALIZE**: 提交上一次升级，目前 Namenode 命令不再支持 “-finalize” 选项，建议用户使用 “hdfs dfsadmin -finalizeUpgrade” 命令进行提交操作。
- **ROLLBACK**: 回滚上一次升级，调用 doRollback()方法执行回滚操作。
- **BOOTSTRAPSTANDBY**: 拷贝 Active Namenode 的最新命名空间数据到 Standby Namenode，调用 BootstrapStandby.run()方法执行操作。
- **INITIALIZESHAREDEDITS**: 初始化 editlog 的共享存储空间，并从 Active Namenode 中拷贝足够的 editlog 数据，使得 Standby 节点能够顺利启动。这里调用了静态方法 initializeSharedEdits()执行操作。
- **BACKUP**: 启动 backup 节点，这里直接构造一个 BackupNode 对象并返回。
- **CHECKPOINT**: 启动 checkpoint 节点，也是直接构造 BackupNode 对象并返回。
- **RECOVER**: 恢复损坏的元数据以及文件系统，这里调用了 doRecovery()方法执行操作。

- **METADATAVERSION**: 确认配置文件夹存在，并且打印 fsimage 文件和文件系统的元数据版本。
- **UPGRADEONLY**: 升级 Namenode，升级完成后关闭 Namenode。
- 默认情况: 其他所有选项的执行都是直接通过 NameNode 的构造方法构造 NameNode 对象，并返回的。

```
public static NameNode createNameNode(String argv[], Configuration conf)
    throws IOException {
    if (conf == null)
        conf = new HdfsConfiguration();
    // 解析命令行的参数
    StartupOption startOpt = parseArguments(argv);
    if (startOpt == null) {
        printUsage(System.err);
        return null;
    }
    setStartupOption(conf, startOpt);

    // 根据启动选项调用对应的方法执行操作
    switch (startOpt) {
        case FORMAT: {
            boolean aborted = format(conf, startOpt.getForceFormat(),
                startOpt.getInteractiveFormat());
            terminate(aborted ? 1 : 0);
            return null; // avoid javac warning
        }
        // ...
        // 在默认情况下直接构造 NameNode 对象并返回
        default: {
            DefaultMetricsSystem.initialize("NameNode");
            return new NameNode(conf);
        }
    }
}
```

NameNode 的构造方法也非常简单，首先解析配置文件，确认当前 Namenode 是否开启了 HA，然后调用 `initialize()` 方法初始化 NameNode 对象，初始化完成后进入 Standby 状态。如果出现异常，则调用 `NameNode.stop()` 方法停止 Namenode 服务。

```
protected NameNode(Configuration conf, NamenodeRole role)
    throws IOException {
    this.conf = conf;
    this.role = role;
    setClientNamenodeAddress(conf);
    String nsId = getNameServiceId(conf);
    String namenodeId = HAUtil.getNameNodeId(conf, nsId);
    this.haEnabled = HAUtil.isHAEnabled(conf, nsId); // 根据配置确认是否开启了 HA
    // 初始启动的 Namenode 都处于 Standby 状态
```

Hadoop 2.X HDFS 源码剖析

```
state = createHARState(getStartupOption(conf));
this.allowStaleStandbyReads = HAUtil.shouldAllowStandbyReads(conf);
this.haContext = createHAContext();
try {
    initializeGenericKeys(conf, nsId, namenodeId);
    initialize(conf); // 调用 initialize() 方法初始化 Namenode
    try {
        haContext.writeLock();
        // 初始化完成后, Namenode 进入 Standby 状态
        state.prepareToEnterState(haContext);
        state.enterState(haContext);
    } finally {
        haContext.writeUnlock();
    }
} catch (IOException e) {
    this.stop(); // 出现异常, 直接停止 Namenode 服务
    throw e;
} catch (HadoopIllegalArgumentException e) {
    this.stop(); // 直接停止 Namenode 服务
    throw e;
}
}
```

`NameNode.initialize()`的实现也不复杂, 它构造 HTTP 服务器, 构造 RPC 服务器, 初始化 `FSNamesystem` 对象, 最后调用 `startCommonServices()`启动 HTTP 服务器、RPC 服务器。

```
protected void initialize(Configuration conf) throws IOException {
    // ...
    // 启动 HTTP 服务
    if (NamenodeRole.NAMENODE == role) {
        startHttpServer(conf);
    }

    // 初始化 FSNamesystem
    loadNamesystem(conf);

    // 创建 RPC 服务
    rpcServer = createRpcServer(conf);

    // ...
    // 构造 JvmPauseMonitor 对象, 并启动
    pauseMonitor = new JvmPauseMonitor(conf);
    pauseMonitor.start();
    metrics.getJvmMetrics().setPauseMonitor(pauseMonitor);

    // 启动 httpServer 以及 rpcServer
    startCommonServices(conf);
}
```

至此, NameNode 就已经成功地启动了, 但是 NameNode 将对文件系统的管理都委托给了 FSNamesystem 对象, NameNode 会调用 FSNamesystem.loadFromDisk() 创建 FSNamesystem 对象。FSNamesystem.loadFromDisk() 首先调用构造方法构造 FSNamesystem 对象, 然后将 fsimage 以及 editlog 文件加载到命名空间中。代码如下:

```
static FSNamesystem loadFromDisk(Configuration conf) throws IOException {
    FSImage fsImage = new FSImage(conf,
        FSNamesystem.getNamespaceDirs(conf),
        FSNamesystem.getNamespaceEditsDirs(conf));
    // 创建 FSNamesystem 对象
    FSNamesystem namesystem = new FSNamesystem(conf, fsImage, false);
    StartupOption startOpt = NameNode.getStartupOption(conf);
    if (startOpt == StartupOption.RECOVER) {
        namesystem.setSafeMode(SafeModeAction.SAFEMODE_ENTER);
    }

    long loadStart = now();
    try {
        // 加载 fsimage 以及 editlog 文件
        namesystem.loadFSImage(startOpt);
    } catch (IOException ioe) {
        fsImage.close();
        throw ioe;
    }
    return namesystem;
}
```

FSNamesystem 的构造方法比较长, 但是逻辑很简单, 主要是从配置文件中获取参数, 然后构造 FSDirectory、BlockManager、SnapshotManager、CacheManager、SafeModeInfo 等对象。需要注意的是, FSNamesystem 的构造方法并不从磁盘上加载 fsimage 以及 editlog 文件, 这些操作是在创建 FSNamesystem 对象成功后, 在 loadFromDisk() 中执行的。如果 FSNamesystem 初始化失败, 则会调用 FSNamesystem.close() 方法关闭 FSNamesystem 启动的所有服务。

```
FSNamesystem(Configuration conf, FSImage fsImage, boolean ignoreRetryCache)
    throws IOException {

    this.fsImage = fsImage;
    try {
        // ...
        this.blockManager = new BlockManager(this, this, conf);
        this.datanodeStatistics = blockManager.getDatanodeManager().getDatanode
Statistics();
        this.blockIdGenerator = new SequentialBlockIdGenerator(this.blockManager);
        // ...
        this.dtSecretManager = createDelegationTokenSecretManager(conf);
        this.dir = new FSDirectory(this, conf);
        this.snapshotManager = new SnapshotManager(dir);
        this.cacheManager = new CacheManager(this, conf, blockManager);
    }
```

```
        this.safeMode = new SafeModeInfo(conf);
        this.auditLoggers = initAuditLoggers(conf);
        this.isDefaultAuditLogger = auditLoggers.size() == 1 &&
            auditLoggers.get(0) instanceof DefaultAuditLogger;
        this.retryCache = ignoreRetryCache ? null : initRetryCache(conf);
        this.nnConf = new NNConf(conf);
    } catch (IOException e) {
        LOG.error(getClass().getSimpleName() + " initialization failed.", e);
        close();
        throw e;
    } catch (RuntimeException re) {
        LOG.error(getClass().getSimpleName() + " initialization failed.", re);
        close();
        throw re;
    }
}
```

成功构造了 FSNamesystem 对象后, NameNode 的构造方法会调用 StandbyState.enterState() 进入 Standby 状态。这部分代码请读者参考上一节的分析。至此, NameNode 也就成功地启动了, 然后管理员就可以通过 DFSHAAdmin 将 Standby Namenode 转换为 Active Namenode 了。

3.7.4 名字节点的停止

名字节点的停止操作非常简单, 在 Namenode.main() 方法中调用了 StringUtils.startupShutdownMessage() 添加一个挂钩, 这个挂钩会在 Namenode 结束运行并且对应的 JVM 虚拟机退出时, 在日志中输出退出信息。startupShutdownMessage() 方法的代码如下:

```
public static void startupShutdownMessage(Class<?> clazz, String[] args,
    final org.apache.commons.logging.Log LOG) {
    final String hostname = NetUtils.getHostname();
    final String classname = clazz.getSimpleName();
    ShutdownHookManager.get().addShutdownHook(
        new Runnable() {
            @Override
            public void run() {
                LOG.info(toStartupShutdownString("SHUTDOWN_MSG: ", new String[]{
                    "Shutting down " + classname + " at " + hostname}));
            }
        }, SHUTDOWN_HOOK_PRIORITY);
}
```


第 4 章 Datanode（数据节点）

在上一章中我们介绍了 HDFS 中最重要的实体 Namenode。这一章我们介绍 HDFS 中另一个非常重要的实体 Datanode，也就是数据节点。

Datanode 以存储数据块（Block）的形式保存 HDFS 文件，同时 Datanode 还会响应 HDFS 客户端读、写数据块的请求。Datanode 会周期性地向 Namenode 上报心跳信息、数据块汇报信息（BlockReport）、缓存数据块汇报信息（CacheReport）以及增量数据块汇报信息。Namenode 会根据块汇报的内容，修改 Namenode 的命名空间（Namespace），同时向 Datanode 返回名字节点指令。Datanode 会响应 Namenode 返回的名字节点指令，如创建、删除和复制数据块指令等。

4.1 Datanode 逻辑结构

在介绍 Datanode 的代码实现前，我们首先介绍 Datanode 的逻辑结构，帮助读者从总体上认识 Datanode。由于 HDFS 2.X 引入了 Federation 架构，造成 Datanode 的代码结构改动非常大，所以本节首先介绍 HDFS Federation 架构，然后给出 Datanode 的逻辑结构。

4.1.1 HDFS 1.X 架构

在介绍 HDFS Federation 架构前，我们先来看一下 HDFS 1.X 的逻辑架构，如图 4-1 所示，HDFS 1.X 架构从逻辑上可以分为两层。

- 命名空间管理层：管理整个文件系统的命名空间（Namespace），包括文件系统目录树中的文件信息、目录信息以及文件包含的数据块信息等。对外提供常见的文件系统操作，例如创建文件、修改文件以及删除文件等。
- 数据块存储管理层：该层分为两个部分。
 - 数据块管理：管理数据节点（Datanode）信息以及数据块信息，对外提供操作数据块的接口，例如创建、删除、修改、获取数据块位置信息等。
 - 存储（Storage）管理：管理 Datanode 上保存数据块的物理存储以及数据块文件，对外提供写数据块文件、读数据块文件以及复制数据块文件等功能。

通过 Namenode（名字节点）章节的学习，我们知道 Namenode 实现了命名空间管理层以

及数据块存储管理层中的数据块管理功能，而 Datanode 则实现了数据块存储管理层中的存储管理部分。如图 4-2 所示，Datanode 会在 Namenode 上注册，并定期向 Namenode 发送数据块汇报与心跳，Namenode 则会通过心跳响应发送数据块操作指令给 Datanode，例如复制、删除以及恢复数据块等指令。可以说 Namenode 的数据块管理层和 Datanode 共同完成了 HDFS 的数据块存储管理功能。

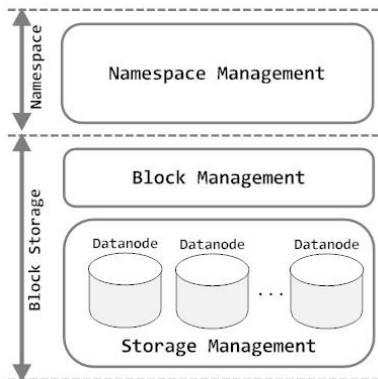


图 4-1 HDFS 1.X 架构示意图

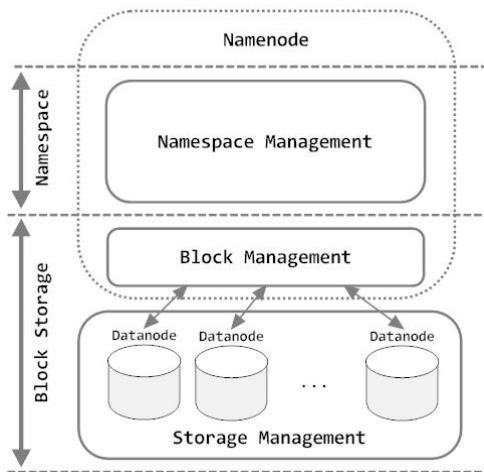


图 4-2 HDFS 1.X 架构实现图

HDFS 1.X 架构使用一个 Namenode 来管理文件系统的命名空间以及数据块信息，这使得 HDFS 的实现非常简单，但是单一的 Namenode 会导致以下缺点。

- 由于 Namenode 在内存中保存整个文件系统的元数据，所以 Namenode 内存的大小直接限制了文件系统的大小。
- 由于 HDFS 文件的读写等流程都涉及与 Namenode 交互，所以文件系统的吞吐量受限于单个 Namenode 的处理能力。
- Namenode 作为文件系统的中心节点，无法做到数据的有效隔离。
- Namenode 是集群中的单一故障点，有可用性隐患。
- Namenode 实现了数据块管理以及命名空间管理功能，造成这两个功能高度耦合，难以让其他服务单独使用数据块存储功能。

考虑到上述缺点，HDFS 2.X 架构提供了 HDFS Federation 功能。

4.1.2 HDFS Federation

为了能够水平扩展 Namenode，HDFS 2.X 提供了 Federation 架构。如图 4-3 所示，Federation 架构的 HDFS 集群可以定义多个 Namenode/Namespace，这些 Namenode 之间是相互独立的，它们各自分工管理着自己的命名空间。而 HDFS 集群中的 Datanode 则提供数据块的共享存储功能，每个 Datanode 都会向集群中所有的 Namenode 注册，且周期性地向所有的 Namenode

发送心跳和块汇报，然后执行 Namenode 通过响应发回的名字节点指令。

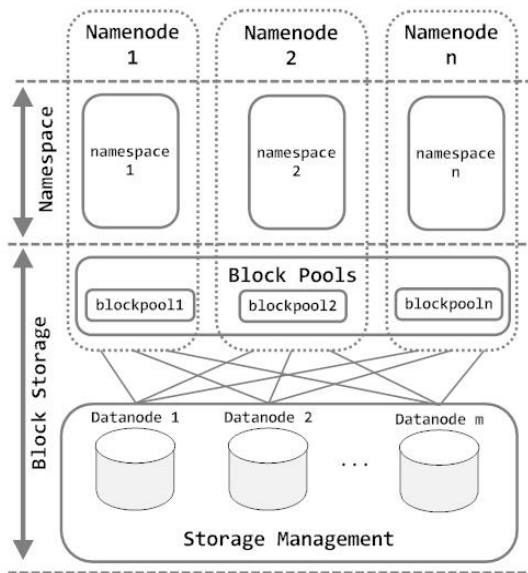


图 4-3 HDFS Federation 架构示意图

HDFS Federation 引入了两个新的概念：块池 (BlockPool) 和命名空间卷 (NamespaceVolume)。

- 块池：一个块池由属于同一个命名空间的所有数据块组成，这个块池中的数据块可以存储在集群中的所有 Datanode 上，而每个 Datanode 都可以存储集群中所有块池的数据块。这里需要注意的是，每个块池都是独立管理的，不会与其他块池交互。所以一个 Namenode 出现故障时，并不会影响集群中的 Datanode 服务于其他的 Namenode。
- 命名空间卷：一个 Namenode 管理的命名空间以及它对应的块池一起被称为命名空间卷，当一个 Namenode/Namespace 被删除后，它对应的块池也会从集群的 Datanode 上删除。需要特别注意的是，当集群升级时，每个命名空间卷都会作为一个基本的单元进行升级。

HDFS Federation 架构相对于 HDFS 1.X 架构具有如下优点。

- 支持 Namenode/Namespace 的水平扩展性，同时为应用程序和用户提供了命名空间卷级别的隔离性。
- Federation 架构实现起来比较简单，Namenode (包括 Namespace) 的实现并不需要太大的改变，只需更改 Datanode 的部分代码即可。例如将 BlockPool 作为数据块存储的一个新层次，以及更改 Datanode 内部的数据结构等。

Federation 的出现为 HDFS 架构提供了新的可能，在未来我们可以将 BlockStorage 抽象成一个新的层次，HDFS Namespace 将使用 Block Storage 层的功能来存储数据块，如图 4-4 所示。

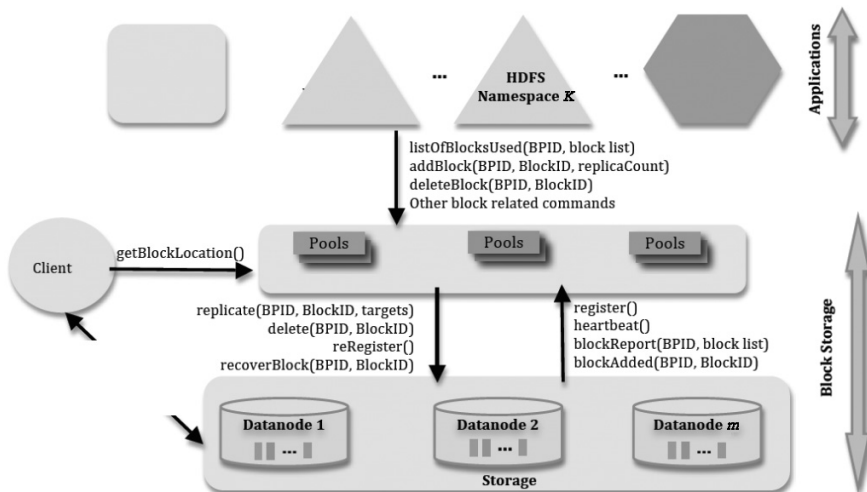


图 4-4 Block Storage 架构示意图

在 BlockStorage 层中，每个块池都是一个独立的数据块集合，块池在管理上与其他块池独立，相互之间不需要协调。Datanode 则提供共享存储功能，存储所有块池的数据块。Datanode 会定期向 BlockStorage 层注册并发送心跳，同时会为每个块池发送块汇报。Block Storage 层会向 Datanode 发送数据块管理命令，之后 Datanode 会执行这些管理命令，例如数据块的复制、删除等。Block Storage 层会对上层应用提供管理数据块的接口，例如在指定块池添加数据块、删除数据块等。

分离出 Block storage 层会带来以下优势。

- 解耦合 Namespace 管理以及 Block Storage 管理。
- 其他应用可以绕过 Namenode/Namesapce 直接管理数据块，例如 HBase 等应用可以直接使用 Block Storage 层。
- 可以在 Block Storage 上构建新的文件系统（non-HDFS）。
- 使用分离的 Block Storage 层为分布式命名空间的实现提供了基础。

4.1.3 Datanode 逻辑结构

在 HDFS Federation 小节中我们介绍了 Federation 架构，为了支持 Federation 架构，对 Datanode 逻辑结构进行了比较大的调整。图 4-5 给出了 Datanode 逻辑结构图，Datanode 从逻辑上可以切分为如下几个模块。

- 数据层：Datanode 的所有服务都是建立在数据块存储功能基础上的，我们将 Datanode 中负责在本地磁盘存储数据块的部分抽象为数据层。数据层包括两个主要的模块。
 - DataStorage——数据存储：负责管理与组织 Datanode 的磁盘存储空间，同时也负责管理存储空间的生命周期（包括升级、回滚、提交等操作）。在 HDFS

Federation 架构中, 一个 Datanode 可以保存多个块池的数据块, HDFS 定义了 BlockPoolSliceStorage 类管理 Datanode 上单个块池的存储空间。DataStorage 类会持有所有 BlockPoolSliceStorage 对象的引用, 并通过这些引用管理 Datanode 上的所有块池。

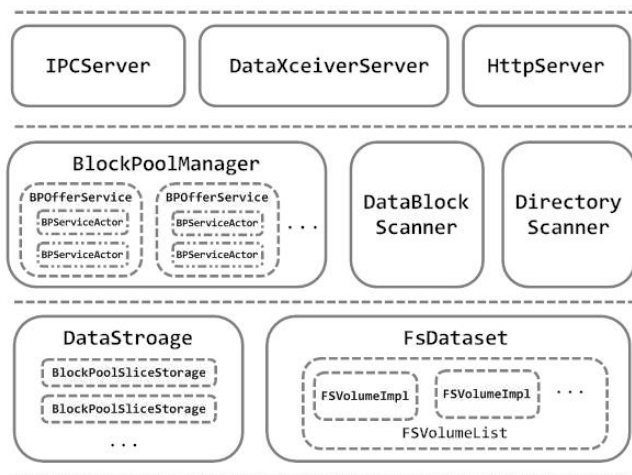


图 4-5 Datanode 逻辑结构图

- **FsDataset**——文件系统数据集：抽象了 Datanode 管理数据块的所有操作，例如创建数据块文件、维护数据块文件和校验和文件的对应关系等。我们知道每个 Datanode 都可以配置若干个不同类型的存储目录来保存数据块（通过配置项 `dfs.datanode.data.dir` 配置），所以 HDFS 定义了 `FSVolumeImpl` 类来管理 Datanode 上单个存储目录保存的所有数据块，同时定义了 `FSVolumeList` 类来维护 Datanode 上所有 `FSVolumeImpl` 对象的引用。`FsDataset` 会通过 `FSVolumeList` 类提供的功能管理 Datanode 上所有存储目录保存的数据块。
- **逻辑层**：在数据层的基础上，Datanode 会执行若干 HDFS 逻辑，例如向 Namenode 汇报数据块存储状态、发送心跳、扫描损坏的数据块等，我们将 Datanode 上负责执行 HDFS 逻辑的部分抽象为逻辑层。逻辑层包括三个主要的模块。
 - **BlockPoolManager**：管理所有块池的接口类。我们知道 HDFS2.X 版本提供了 Federation 机制，也就是每个 HDFS 集群都可以配置多个命名空间，每个命名空间在 Datanode 上都有一个与之对应的块池。一个 `BlockPoolManager` 对象会持有多个 `BPOfferService` 对象的实例，每个 `BPOfferService` 对象都管理这个 Datanode 的一个块池。HDFS2.X 中除了引入 Federation 机制外，还引入了 HA 机制，每个命名空间都可以定义两个 Namenode，一个作为 Active Namenode，一个作为 Standby Namenode。所以每个 `BPOfferService` 对象又都会持有两个 `BPServiceActor` 对象，一个 `BPServiceActor` 对象对应于命名空间中的一个 Namenode，该对象负责向这个 Namenode 发送心跳、块汇报、缓存汇报以及增量块汇报，并执行 Namenode 返回的名字节点指令（`DatanodeCommand`）。

- **DataBlockScanner**: 一个独立的线程，周期性地扫描每个数据块并检查数据块的校验和是否正常。
- **DirectoryScanner**: 一个独立的线程，定时发起对磁盘数据块的扫描，对比内存中元数据与实际磁盘存储数据块的差异，然后更新内存中的元数据，使之与磁盘保存的数据块信息一致。
- **服务层**: 除了上面介绍的数据层以及逻辑层外，**Datanode** 还包括一个对外服务层，服务层中的模块用于支持其他节点与 **Datanode** 通信，以及访问 **Datanode** 状态等功能。服务层包括三个模块。
 - **HttpServer**: 对外提供 HTTP 服务，可用于展示 **Datanode** 内部状态。
 - **IPCServer**: RPC 服务端，响应来自于 **Client**、**Namenode** 以及其他 **Datanode** 的 RPC 请求。
 - **DataXceiverServer**: 数据传输服务端，响应来自于 **Client** 以及其他 **Datanode** 的流式接口请求。

了解了 **Datanode** 的整体逻辑结构之后，我们在后面的章节中就逐一介绍各个模块。

4.2 Datanode 存储

我们知道，**Datanode** 和 **Namenode** 存储的 HDFS 数据都是在 Linux 磁盘上持久化的，且无论 **Datanode** 还是 **Namenode**，都会在磁盘上维护一定的文件目录结构，以方便进行数据的组织以及版本管理等操作。在 **Datanode** 的源码中，**DataStorage** 类提供了管理与组织 **Datanode** 磁盘存储空间，以及管理 **Datanode** 存储空间生命周期（包括支持升级、回滚、提交等操作，维护存储空间的状态机等）等功能。

本节就介绍 **DataStorage** 类的实现。首先介绍 **Datanode** 升级机制，然后介绍 **Datanode** 磁盘存储结构，最后介绍 **DataStorage** 类的代码实现。

4.2.1 Datanode 升级机制

DataStorage 最重要的功能之一是管理磁盘存储空间的生命周期。升级是磁盘存储空间生命周期管理中最重要的一环，尤其是对于 HDFS 这样的分布式存储系统，升级需要重点考虑以下几个问题。

- **版本兼容性问题**: 不同版本之间兼容性的设计，高版本是否需要兼容低版本？不同组件之间，例如 **Datanode** 和 **Namenode** 之间的版本是否需要一致？
- **升级消耗的磁盘空间问题**: 我们知道 **Datanode** 最主要的功能是存储数据块，而这些数据块往往占用比较大的磁盘空间。如何做到既能保留原有版本的数据，正确地完成升级操作，同时又能节省磁盘空间呢？
- **回滚要求**: 在 HDFS 的升级过程中，很有可能出现各种异常，包括硬件的异常、软件的异常以及人为的操作失误。那么如何在升级失败的情况下，回滚到之前正确的

状态呢？

考虑到上面的问题，HDFS 实现了一套完整的升级机制。

- 关于兼容性：HDFS 支持节点版本的向上升级，但并不支持向下降版本。HDFS 各个组件作为一个整体升级或者回滚，同一集群中有不同版本的组件是不被支持的。同时 HDFS 确保在升级过程中，旧版本的结构可以迁移至新版本。这里要注意，如果新版本的文件系统布局不需要改变，则升级非常简单，只需导入新的代码与依赖包，加载配置文件，重启服务即可；如果需要改变文件系统布局、元数据数据结构等 (layoutVersion 发生改变)，则升级过程会比较复杂，需要考虑回滚等容错机制。
- 关于回滚：HDFS 回滚机制主要是通过备份旧版本数据实现的，回滚时将旧版本数据复制到原有目录中即可。HDFS 仅保留前一个版本的数据，同时引入升级提交机制，当管理员提交了一次升级时（通过“`hadoop dfsadmin -finalizeUpgrade`”命令），HDFS 将会删除之前的版本，也就是提交升级之后，无法再进行回滚操作。
- 关于磁盘空间：在 HDFS 的升级过程中，需要复制低版本数据以支持回滚操作。但由于 Datanode 存储的数据块占用的磁盘空间往往比较大，所以数据节点使用了 Linux 硬链接方式，也就是将新版本和旧版本中两个数据块文件的引用指向磁盘中的同一个数据块，巧妙地节省了磁盘空间。

HDFS 升级机制是通过磁盘文件目录的配合来进行的，如图 4-6 所示，我们可以将 Datanode 升级分为三个步骤：升级、回滚和提交。

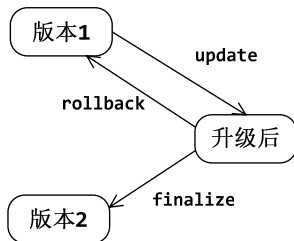


图 4-6 Datanode 升级示意图

1. 升级操作

升级操作就是将 Datanode 从低版本升级到高版本。主要涉及以下目录。

- `current` 目录：保存当前版本数据的目录。
- `previous.tmp` 目录：在升级过程中，保存当前版本数据的目录。
- `previous` 目录：升级后，保存上一版本数据的目录。

在 Datanode 升级时，会将当前版本数据所在的 `current` 目录改名为 `previous.tmp`，然后为新版本数据重建 `current` 目录（包括重建新版本的 `VERSION` 文件、新的子目录结构）。接下来建立 `current` 目录与 `previous.tmp` 中数据块文件和校验和文件之间的硬链接。最后将 `previous.tmp` 改名为 `previous`，完成升级。这时文件夹中的 `current` 目录存储的是当前版本的数据，`previous` 目录保存的是上一版本的数据。

2. 回滚操作

回滚操作就是将 Datanode 回滚至上一版本。主要涉及以下目录。

- **previous** 目录：升级后，保存上一版本数据的目录。
- **removed.tmp** 目录：在回滚过程中，保存当前版本数据的目录。
- **current** 目录：回滚后，保存当前版本数据的目录。

在 Datanode 回滚时，会先将 **current** 目录改名为 **removed.tmp**，然后将 **previous** 目录改名为 **current**，最后删除 **removed.tmp** 目录。回滚完成后，文件夹中只有 **current** 一个目录保存回滚后版本的数据。

3. 升级提交操作

升级提交操作就是提交当前升级，删除上一版本数据，提交之后，Datanode 将不可回滚。主要涉及以下目录。

- **previous** 目录：升级后，保存上一版本数据的目录。
- **finalized.tmp** 目录：在提交过程中，保存前一版本数据的目录。

在升级提交时，先将 **previous** 目录改名为 **finalized.tmp**，然后删除 **finalized.tmp** 目录。提交操作完成后，文件夹中只有 **current** 一个目录保存当前版本的数据。

4. 临时状态

我们注意到，Datanode 无论是升级、回滚还是提交操作，都没有一步完成，而是将中间的临时状态都保留下来，如图 4-7 所示。这样做的目的是，在发生故障时可以恢复到操作前的状态。

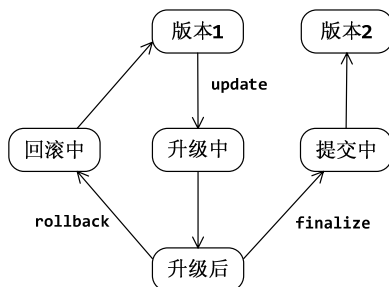


图 4-7 临时状态示意图

我们以升级操作为例，如果不使用 **previous.tmp** 目录，而是直接将新建的 **current** 目录硬链接到 **previous** 目录，若这个过程发生错误，那么 **current** 与 **previous** 目录之间的数据就会不一致，导致升级失败。这时，当数据节点再次启动时，它并不能根据 **current** 和 **previous** 目录的不一致判断升级过程中是否发生了错误。而引入临时目录 **previous.tmp** 后，数据节点启动时如果发现存储空间中有该临时目录，就可以判断上一次升级失败了。回滚操作中的临时目录 **removed.tmp** 以及提交操作中的临时目录 **finalized.tmp** 作用也是相同的。

4.2.2 Datanode 磁盘存储结构

尽管 Datanode 并不保存 HDFS 文件和目录的元数据,但是 Datanode 还是需要保存一部分 Datanode 自身的元数据的,这些元数据是通过 Datanode 磁盘存储上的一些文件和目录来保存的,所以这一节我们就先学习 Datanode 磁盘存储结构。

Datanode 可以定义多个存储目录保存数据块,如下配置所示,Datanode 定义了“/dfs/data”和“/dfs/data2”两个数据存储目录,这两个存储目录共同组成了 Datanode 的存储空间。这里要注意,Datanode 的多个存储目录存储的数据块并不相同,并且不同的存储目录可以是异构的,这样的设计可以提高数据块 IO 的吞吐率。

```
<property>
  <name>dfs.data.dir</name>
  <value> /dfs/data, /dfs/data2</value>
</property>
```

图 4-8 给出了 Datanode 存储目录的树形结构,当前 Datanode 定义了两个存储目录:“/dfs/data”和“/dfs/data2”。在每个存储目录中,又包含了子目录 current 保存当前版本的数据,子目录 previous 保存上一版本的数据(请参考 Datanode 升级机制小节)。

由于 HDFS 2.X 版本引入了 Federation 机制(请参考 HDFS Federation 小节),所以 Datanode 会保存多个块池的数据块,这些块池的数据块会分布在 Datanode 定义的所有存储目录下。假设当前 Datanode 保存了两个块池的数据块,分别是块池 1 和块池 2,那么这两个块池在 Datanode 的每个存储目录下都会有单独的块池目录保存数据块。如图 4-9 所示,data 和 data2 目录是 Datanode 定义的存储目录,而它们的子目录 blockpool1 和 blockpool2 则是块池 1 和块池 2 在存储目录下的块池目录。

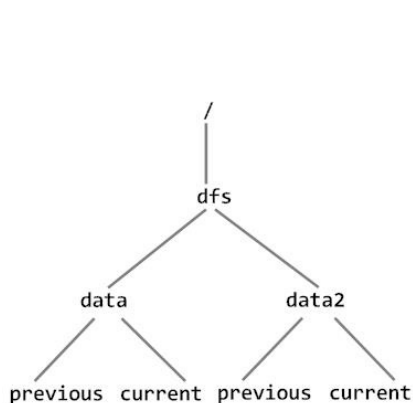


图 4-8 存储目录

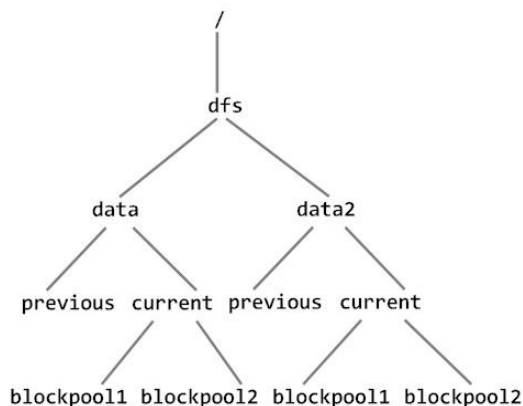


图 4-9 存储目录以及块池目录

图 4-9 给出的目录结构不适用于独立的 Namenode 升级操作,假设 blockpool1 对应的 Namenode1 触发了升级操作,那么 blockpool2 块池目录也需要一同升级,因为它们都在同一个存储目录下的 current 文件夹中。图 4-10 给出了当前 Datanode 的磁盘目录结构,无论是在

存储目录中还是块池目录中都包含了支持升级操作的 **current** 和 **previous** 文件夹。这种结构使得各个快照对应的 **Namenode** 可以独立升级(通过块池目录的升级文件夹),同时如果 **Datanode** 升级与 **Namenode** 升级解耦合,这种目录结构也可以支持 **Datanode** 级别的升级操作。

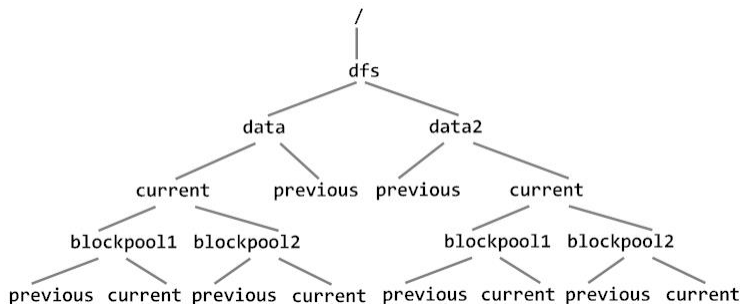


图 4-10 存储目录与块池目录结构

了解了存储目录与块池目录的关系后,我们再来看它们在磁盘上的真实结构。图 4-11 给出了在 **Datanode** 配置的一个存储目录下运行 “tree” 命令的结果。

```

data/dfs/data/
├── current
│   ├── BP-1079595417-192.168.2.45-1412613236271
│   │   ├── current
│   │   │   ├── VERSION
│   │   │   ├── finalized
│   │   │   ├── subdir0
│   │   │   ├── subdir1
│   │   │   ├── blk_1073741825
│   │   │   ├── blk_1073741825_1001.meta
│   │   │   ├── lazyPersist
│   │   │   ├── rbw
│   │   │   ├── dncp_block_verification.log.curr
│   │   │   ├── dncp_block_verification.log.prev
│   │   │   ├── tmp
│   │   │   ├── VERSION
│   │   │   └── in_use.lock

```

图 4-11 Datanode 磁盘存储结构

- **BP-1079595417-192.168.2.45-1412613236271**: 这个目录是一个块池目录,块池目录保存了一个块池在当前存储目录下存储的所有数据块,在 **Federation** 部署方式中,**Datanode** 的一个存储目录会包含多个以 “BP” 开头的块池目录。BP 后面会紧跟一个唯一的随机块池 ID,在这个示例中就是 1079595417。接下来的 IP 地址 192.168.2.45 是当前块池对应的 **Namenode** 的 IP 地址。最后一个部分是这个块池的创建时间。
- **VERSION 文件**: 和 **Namenode** 以及 **JournalNode** 的 **VERSION** 文件类似,块池目录的 **VERSION** 文件同样包含了文件系统布局版本 (layoutVersion)、HDFS 集群 ID (clusterId) 以及创建时间 (cTime) 等集群信息。除此之外,块池目录的 **VERSION**

文件还包含了以下信息。

- storageType——存储类型，在这个示例中，存储类型被设置为 `Datanode`。
- blockpoolID——块池 ID，这个字段的值与块池目录下保存的块池 ID 是一样的。
- **finalized/rbw**: `finalized` 和 `rbw` 目录都是用于存储数据块的，包括数据块文件以及对应的校验和文件。`rbw` (`replica being written`，正在写入副本) 目录保存了正在由 HDFS 客户端写入当前 `Datanode` 的数据块。`finalized` 目录包含了已经完成写入操作的数据块，由于这样的数据块可能非常多，所以 `finalized` 目录会以特定的目录结构存储这些数据块。
- **lazyPersist**: HDFS 2.X 中引入了一个新的特性，用于支持将临时数据写入内存，然后通过懒持久化 (`lazyPersist`) 方式写入磁盘。如果用户开启了这个特性，`lazyPersist` 目录就用于将内存中的临时数据懒持久化到磁盘。
- **dncp_block_verification.log**: 这个文件用于记录 `Datanode` 最近一次确认所有数据块的内容和校验值匹配的时间。`Datanode` 会以升序按照这个最后确认的时间安排后台的数据块确认操作。这个文件会定期滚动，所以会存在 `curr` 和 `prev` 文件。
- **in_use.lock**: 这是一个被 `Datanode` 线程持有的锁文件，用于防止多个 `Datanode` 线程启动并且并发修改这个存储目录。

4.2.3 DataStorage 实现

Datanode 最重要的功能就是管理磁盘上存储的 HDFS 数据块。Datanode 将这个管理功能切分为两个部分：① 管理与组织磁盘存储目录（由 `dfs.data.dir` 指定），如 `current`、`previous`、`detach`、`tmp` 等，这个功能由 `DataStorage` 类实现；② 管理与组织数据块及其元数据文件，这个功能主要由 `FsDatasetImpl` 相关类实现。本节介绍 `DataStorage` 类的实现。

1. Storage 类继承关系

DataSource 的父类为 Storage，图 4-12 给出了 Storage 类的继承关系。StorageInfo 为根接口，描述存储的基本信息。子类 Storage 是抽象类，它为 Datanode、Namenode 提供抽象的存储服务。

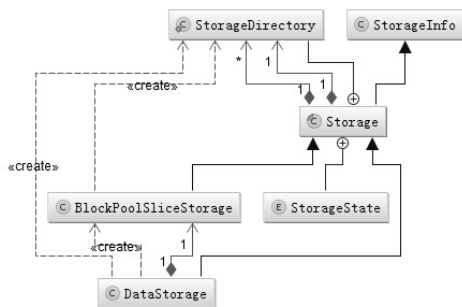


图 4-12 Storage 类继承关系图


一个 `Storage` 可以定义多个存储目录,存储目录由 `Storage` 的内部类 `StorageDirectory` 描述, `StorageDirectory` 类定义了存储目录上的通用操作。这里我们以 `Datanode` 的配置为例, `Datanode` 可以定义多个存储目录保存数据块,如下配置所示, `Datanode` 定义了两个数据存储目录,即“/data/hdfs/dfs/data”和“/data/hdfs/dfs/data2”。HDFS 会使用一个 `DataStorage` 对象管理整个 `Datanode` 的存储,而这两个存储目录则由两个 `StorageDirectory` 对象管理。

```
<property>
  <name>dfs.data.dir</name>
  <value>/data/hdfs/dfs/data,/data/hdfs/dfs/data2</value>
</property>
```

由于 HDFS 2.X 版本引入了 Federation 机制(请参考 HDFS Federation 小节), `Datanode` 会为多个块池保存数据块, HDFS 定义了 `BlockPoolSliceStorage` 类来管理 `Datanode` 上的一个块池,这个块池分布在 `Datanode` 配置的所有存储目录中。 `DataStorage` 类会持有所有 `BlockPoolSliceStorage` 对象的引用,并通过这些引用管理 `Datanode` 上的所有块池。

2. StorageInfo

`StorageInfo` 用于描述存储的基本信息,如图 4-13 所示,这个类有以下 5 个字段。



StorageInfo	
layoutVersion	int
namespaceID	int
clusterID	String
cTime	long
storageType	NodeType

图 4-13 StorageInfo 字段

- **layoutVersion**: 存储系统布局版本号,当节点存储的目录结构发生改变或者 `fsimage` 和 `editlog` 的格式发生改变时,存储系统布局版本号会更新。这个版本号一般是负数。
- **namespaceID**: 存储系统命名空间标识。
- **cTime**: 存储系统创建时间。
- **clusterID**: 存储系统的集群 ID。
- **storageType**: 节点类型,有 `DATA_NODE`、`NAME_NODE`、`JOURNAL_NODE` 等类型。

这里要注意, `StorageInfo` 中定义的信息都存储在存储目录的 `VERSION` 文件中,一个 `VERSION` 文件就是一个典型的 `Java Properties` (属性)文件,除了上述 5 个属性外,不同类型节点的 `VERSION` 文件还存储了其他特有的属性。

`StorageInfo` 的方法大多是 `get/set` 方法,以及从 `Properties` 文件中读取属性然后赋值字段的方法,比较简单,这里就不再赘述了。

3. Storage.StorageState

`Storage` 类定义了一个非常重要的内部枚举类——`StorageState`,这个枚举类完整地定义了存储空间可能出现的所有状态。在升级、回滚、升级提交、检查点等操作中,节点(`Datanode`

或者 Namenode) 的存储空间可能出现各种异常, 例如误操作、断电、宕机等情况, 这个时候存储空间就可能处于某种中间状态, 引入中间状态, 有利于 HDFS 从错误中恢复过来。对于存储状态的确定, 是在 `StorageDirectory.analyseStorage()` 方法中进行的, 我们在下一节 `StorageDirectory` 中介绍这个方法。

```
public enum StorageState {
    NON_EXISTENT,        // 存储不存在
    NOT_FORMATTED,       // 未格式化
    COMPLETE_UPGRADE,    // 升级完成状态
    RECOVER_UPGRADE,     // 恢复升级状态
    COMPLETE_FINALIZE,   // 完成升级提交
    COMPLETE_ROLLBACK,   // 完成回滚操作
    RECOVER_ROLLBACK,    // 恢复回滚
    COMPLETE_CHECKPOINT, // 完成检查点操作
    RECOVER_CHECKPOINT,  // 恢复检查点操作
    NORMAL;              // 正常状态
}
```

这里要特别注意, `Storage` 状态还与启动选项有关, 这些选项的存储在 `HdfsServerConstants` 类中, 代码如下:

```
static public enum StartupOption{
    FORMAT ("-format"),
    CLUSTERID ("-clusterid"),
    GENCLUSTERID ("-genclusterid"),
    REGULAR ("-regular"),
    BACKUP ("-backup"),
    CHECKPOINT ("-checkpoint"),
    UPGRADE ("-upgrade"),
    ROLLBACK ("-rollback"),
    FINALIZE ("-finalize"),
    ROLLINGUPGRADE ("-rollingUpgrade"),
    IMPORT ("-importCheckpoint"),
    BOOTSTRAPSTANDBY ("-bootstrapStandby"),
    INITIALIZESHAREDEDITS ("-initializeSharedEdits"),
    RECOVER ("-recover"),
    FORCE ("-force"),
    NONINTERACTIVE ("-nonInteractive"),
    RENAMERESERVED ("-renameReserved"),
    METADATAVERSION ("-metadataVersion"),
    UPGRADEONLY ("-upgradeOnly"),
    HOTSWAP ("-hotswap");
    // ...
}
```

在启动选项中, 与 `Storage` 状态相关的有以下几项。

- **FORMAT**: 格式化操作。
- **REGULAR**: 正常启动。

- UPGRADE: 升级操作。
- ROLLBACK: 回滚操作。
- FINALIZE: 提交升级操作。

4. Storage.StorageDirectory

我们知道 Datanode 和 Namenode 都可以定义多个存储目录来存储数据，StorageDirectory 是 Storage 的内部类，定义了管理存储目录的通用方法。

如下代码所示，StorageDirectory 有 5 个字段。

```
final File root;  
final boolean isShared;  
final StorageDirType dirType;  
FileLock lock;  
private String storageUuid = null;
```

- root: 存储目录的根，就是 java.io.File 文件。
- dirType: 当前存储目录的类型。
- isShared: 指示当前目录是否是共享的。例如在 HA 部署中，不同的 Namenode 之间共享存储目录，或者在 Federation 部署中不同的块池之间共享存储目录。
- lock: 独占锁，java.nio.FileLock 类型，用来支持 Datanode 或者 Namenode 线程独占存储目录的锁操作。
- storageUuid: 存储目录的标识符。

StorageDirectory 的方法主要分为三类：获取文件夹相关操作、加锁/解锁操作、存储状态恢复操作。下面我们依次看一下这三种类型方法的实现。

(1) 文件夹操作

获取当前存储目录结构中的各个文件/文件夹的方法，在 HDFS 升级过程中涉及的所有目录都可以通过 StorageDirectory 提供的方法获得：

- getCurrentDir()——获取 current 目录。
- getVersionFile()——获取 current 目录下的 VERSION 文件。
- getPreviousDir()——获取 previous 目录。
- getPreviousVersionFile()——获取 previous 目录下的 VERSION 文件。
- getPreviousTmp()——获取 previous.tmp 目录。
- getRemovedTmp()——获取 removed.tmp 目录。
- getFinalizedTmp()——获取 finalized.tmp 文件。
- getLastCheckpointTmp()——获取 lastcheckpoint.tmp 文件。
- getPreviousCheckpoint()——获取 previous.checkpoint 文件。

(2) 加锁/解锁操作

在 Datanode 磁盘存储结构小节中，我们介绍了存储目录下会有一个 in_use.lock 文件，这个文件用于对当前存储目录加锁，以保证 Datanode 进程对存储目录的独占使用。当 Datanode

进程退出执行时，`in_use.lock` 文件会被删除。`StorageDirectory` 提供了 `tryLock()` 与 `unlock()` 两个锁方法，分别实现了对存储目录加锁以及解锁的功能。

tryLock()

`StorageDirectory` 中真正进行加锁操作的是 `tryLock()` 方法。`tryLock()` 方法会首先构造锁文件，然后调用 `file.getChannel.lock()` 方法尝试获得存储目录的独占锁，如果已经有进程占有锁文件，那么 `file.getChannel.lock()` 就会返回一个 `null` 的引用，表明有另一个节点运行在当前的存储目录上，`tryLock()` 方法会抛出异常并退出执行。如果加锁成功，`tryLock()` 方法会在锁文件中写入虚拟机信息。

加锁成功后，`tryLock()` 方法会调用 `deleteOnExit()` 方法，在 Java 虚拟机运行结束时删除 `in_use.lock` 文件。`tryLock()` 方法的代码如下：

```
FileLock tryLock() throws IOException {
    boolean deletionHookAdded = false;
    // 构造 in_use.lock 文件
    File lockF = new File(root, STORAGE_FILE_LOCK);
    if (!lockF.exists()) { // 锁文件构造失败，则退出执行
        lockF.deleteOnExit();
        deletionHookAdded = true;
    }
    RandomAccessFile file = new RandomAccessFile(lockF, "rws");
    String jvmName = ManagementFactory.getRuntimeMXBean().getName();
    FileLock res = null;
    try {
        // 尝试在锁文件上加锁
        res = file.getChannel().tryLock();
        // 已经有程序获得了锁，那么直接抛出异常
        if (null == res) {
            throw new OverlappingFileLockException();
        }
        // 加锁成功，在锁文件中写入虚拟机信息
        file.write(jvmName.getBytes(Charsets.UTF_8));
    } catch (OverlappingFileLockException oe) {
        // 已经有程序获得了锁，则关闭锁文件，返回 null
        file.close();
        return null;
    } catch (IOException e) {
        // 读取锁文件失败，则关闭锁文件，抛出异常
        file.close();
        throw e;
    }
    if (res != null && !deletionHookAdded) {
        // 加锁成功，在虚拟机运行结束后，删除锁文件
        lockF.deleteOnExit();
    }
    return res;
}
```

unlock()

节点退出时释放锁，关闭 channel。这个方法的实现比较简单，这里就不再贴出代码了。

(3) 存储状态恢复操作

Datanode 在执行升级、回滚、提交操作的过程中会出现各种异常，例如误操作、断电、宕机等情况。那么 Datanode 在重启时该如何恢复上一次中断的操作呢？StorageDirectory 提供了 doRecover()和 analyzeStorage()两个方法，Datanode 会首先调用 analyzeStorage()方法分析当前节点的存储状态，然后根据分析所得的存储状态调用 doRecover()方法执行恢复操作。图 4-14 给出了存储状态恢复操作流程图。

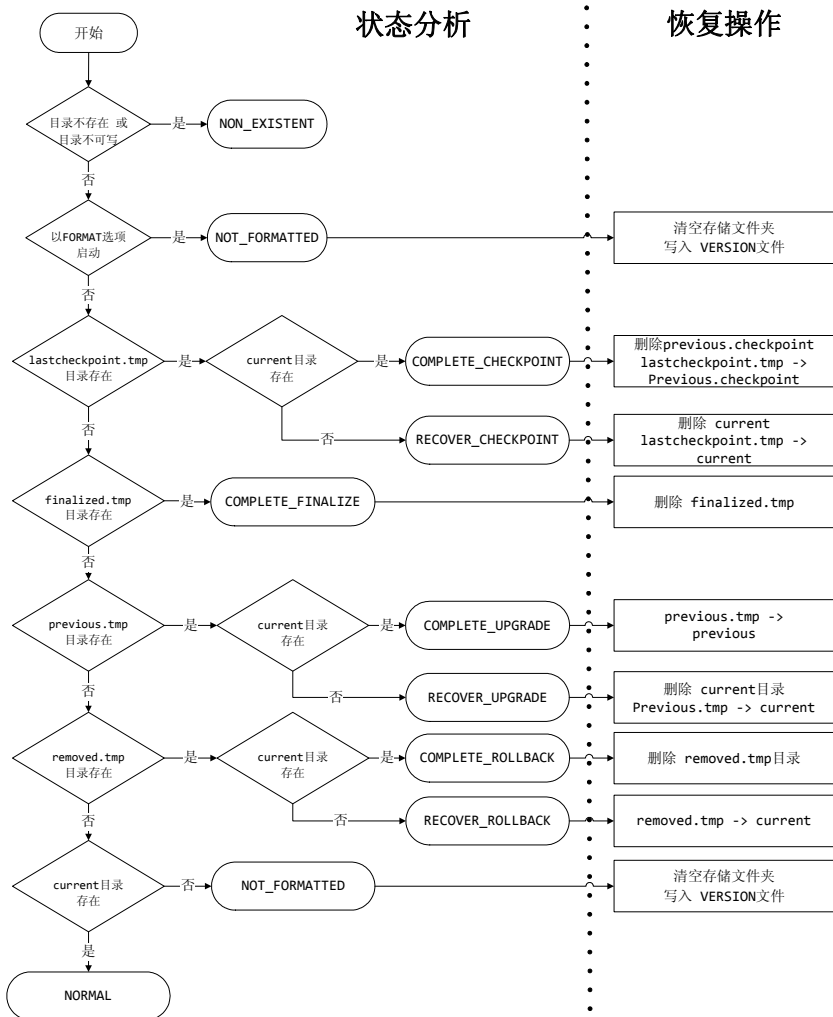


图 4-14 存储状态恢复操作流程图

analyseStorage()

analyseStorage()方法用于在Datanode启动时分析当前Datanode存储目录的状态，Datanode存储目录的状态定义在Storage.StorageState类中，我们在前面的小节中已经介绍了。对存储目录状态的分析还需要结合Datanode的升级机制以及Datanode的启动选项，存储目录状态判断的逻辑如下。

- **NON_EXISTENT**: 以非FORMAT选项启动时，目录不存在；或者目录不可写、路径为文件时，存储目录状态都为NOT_EXISTENT状态。
- **NOT_FORMATTED**: 以FORMAT选项启动时，都为NOT_FORMATTED状态。
- **NORMAL**: 没有tmp中间状态文件夹，则存储目录为正常状态。
- **COMPLETE_UPGRADE**: 存在current/VERSION文件，存在previous.tmp文件夹，则存储目录为升级完成状态。
- **RECOVER_UPGRADE**: 存在previous.tmp文件夹，不存在current/VERSION文件，存储目录应该从升级中恢复。
- **COMPLETE_ROLLBACK**: 存在removed.tmp文件夹，也存在current/VERSION文件，则存储目录的回滚操作成功完成。
- **RECOVER_ROLLBACK**: 存在removed.tmp文件夹，不存在current/VERSION文件，存储目录应该从回滚中恢复。
- **COMPLETE_FINALIZE**: 存在finalized.tmp文件夹，存储目录可以继续执行提交操作。

doRecover()

调用analyseStorage()之后，Datanode就可以确定存储目录的状态了。对于异常状态，可以通过调用doRecover()方法进行恢复，使存储空间的状态恢复到NORMAL状态。

doRecover()的流程如图4-14所示，这里我们以COMPLETE_ROLLBACK和RECOVER_ROLLBACK为例。如果当前状态是COMPLETE_ROLLBACK，只需将removed.tmp目录删除即可将存储目录状态恢复为NORMAL状态；如果当前状态是RECOVER_ROLLBACK，只需将removed.tmp改名为current即可将存储目录状态恢复为NORMAL状态。其他状态同理，请参考图4-14所示操作。这个方法的代码也比较简单，我们就不再赘述了。

5. Storage

Storage是一个抽象类，为Datanode、Namenode提供抽象的存储服务。Storage类管理着当前节点上（可以是Datanode或者Namenode）所有的存储目录，每个存储目录都由一个StorageDirectory对象管理，Storage用一个线性表字段storageDirs存储它管理的所有StorageDirectory，并通过DirIterator迭代器进行遍历。

```
public abstract class Storage extends StorageInfo {
    protected List<StorageDirectory> storageDirs = new ArrayList<StorageDirectory>();
    private class DirIterator implements Iterator<StorageDirectory> {
        //...
    }
}
```

Storage 的方法都比较简单，都是在 StorageDirectory 的线性表或者 VERSION 文件上操作的，这里就不再赘述了。

6. DataStorage

DataStorage 继承自 Storage 抽象类，提供了管理 Datanode 存储空间的功能。本节介绍 DataStorage 类的实现。

在 HDFS Federation 架构中，一个 Datanode 可以保存多个命名空间的数据块，每个命名空间在 Datanode 磁盘上都拥有一个独立的块池（BlockPool），这个块池会分布在 Datanode 的所有存储目录下，它们共同保存了这个块池在当前 Datanode 上的所有数据块。HDFS 定义了 BlockPoolSliceStorage 类管理 Datanode 上单个块池的存储空间（BlockPoolSliceStorage 的实现我们在下一节中介绍），DataStorage 类则定义了 bpStorageMap 字段保存 Datanode 上所有块池 BlockPoolSliceStorage 对象的引用。如下代码所示，bpStorageMap 字段是 Map 类型的，维护了 bpId -> BlockPoolSliceStorage 的映射关系。

```
private final Map<String, BlockPoolSliceStorage> bpStorageMap
    = Collections.synchronizedMap(new HashMap<String, BlockPoolSliceStorage>());
```

Datanode 在启动时会调用 DataStorage 提供的方法初始化 Datanode 的存储空间，在 HDFS Federation 架构中，Datanode 会保存多个命名空间的数据块。对于每一个命名空间，Datanode 都会构造一个 BPOfferService 类维护与这个命名空间 Namenode 的通信（请参考文件系统数据块的 BlockManager 节的 BPOfferService 小节）。如图 4-15 所示，当 BPOfferService 中的 BPServiceActor 类与该命名空间的 Namenode 握手成功后，就会调用 DataNode.initBlockPool() 初始化该命名空间的块池。DataNode.initBlockPool() 方法最终会调用 DataStorage.recoverTransitionRead() 来执行块池存储的初始化操作。本节我们就以 recoverTransitionRead() 作为入口方法来分析 DataStorage 的代码。

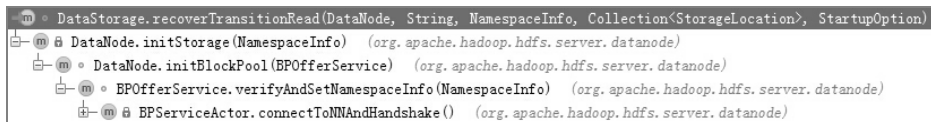


图 4-15 DataStorage 初始化流程

(1) recoverTransitionRead()

recoverTransitionRead() 方法用于在所有存储目录下初始化指定块池。recoverTransitionRead() 方法的执行可以分为以下几个步骤。

- 首先调用重载的 recoverTransitionRead() 方法初始化 Datanode 存储，也就是确保 Datanode 定义的所有存储目录的状态正常，然后在 Datanode 存储层面执行启动参数对应的操作，如备份/升级/回滚/恢复/提交等（请参考 Storage.StorageState 小节）。
- 接下来调用 makeBlockPoolDataDir() 方法在 Datanode 定义的所有存储目录下创建当前命名空间对应块池的块池存储目录。块池存储目录的结构请参考 Datanode 磁盘存储结构小节。

- 最后为块池构造 `BlockPoolSliceStorage` 对象, 并在该对象上调用 `BlockPoolSliceStorage.recoverTransitionRead()` 方法完成块池目录的初始化, 同时在块池存储层面执行启动参数对应的操作(备份/升级/回滚/恢复/提交)。然后将该 `BlockPoolSliceStorage` 对象放入 `DataStorage.bpStorageMap` 字段中保存。

`recoverTransitionRead()` 方法的代码如下:

```
void recoverTransitionRead(DataNode datanode, String bpID, NamespaceInfo nsInfo,
    Collection<StorageLocation> dataDirs, StartupOption startOpt) throws IOException {
    // 首先确保 Datanode 定义的所有存储目录的状态正常, 然后触发启动参数对应的方法
    recoverTransitionRead(datanode, nsInfo, dataDirs, startOpt);

    // 在 Datanode 的所有存储目录下创建块池存储目录, 将所有块池目录放入 bpDataDirs 中保存
    Collection<File> bpDataDirs = new ArrayList<File>();
    for(StorageLocation dir : dataDirs) {
        File dnRoot = dir.getFile();
        File bpRoot = BlockPoolSliceStorage.getBpRoot(bpID, new File(dnRoot,
            STORAGE_DIR_CURRENT));
        bpDataDirs.add(bpRoot);
    }
    makeBlockPoolDataDir(bpDataDirs, null);

    // 构建当前 bpId 对应的 BlockPoolSliceStorage 对象
    BlockPoolSliceStorage bpStorage = new BlockPoolSliceStorage(
        nsInfo.getNamespaceID(), bpID, nsInfo.getCTime(), nsInfo.getClusterID());
    // 在 BlockPoolSliceStorage 对象上调用 recoverTransitionRead() 方法, 执行块池目录的初始化
    bpStorage.recoverTransitionRead(datanode, nsInfo, bpDataDirs, startOpt);
    // 将新构建的 BlockPoolSliceStorage 对象放入 DataStorage.bpStorageMap 中
    addBlockPoolStorage(bpID, bpStorage);
}
```

`recoverTransitionRead()` 方法调用了同名的重载方法 `recoverTransitionRead()` 初始化 `Datanode` 存储空间, 重载的 `recoverTransitionRead()` 方法会调用 `addStorageLocations()` 方法对 `Datanode` 存储空间进行初始化, 然后将 `DataStorage.initialized` 设置为 `true`。重载的 `recoverTransitionRead()` 方法的代码如下:

```
synchronized void recoverTransitionRead(DataNode datanode,
    NamespaceInfo nsInfo, Collection<StorageLocation> dataDirs,
    StartupOption startOpt)
    throws IOException {
    if (initialized) {
        return; // Datanode 已经初始化了, 则没有必要重复执行, 直接返回
    }
    this.storageDirs = new ArrayList<StorageDirectory>(dataDirs.size());
    // 调用 addStorageLocations() 方法对 Datanode 存储空间进行初始化
    addStorageLocations(datanode, nsInfo, dataDirs, startOpt, true, false);

    this.initialized = true; // 设置 Datanode 的初始化状态
}
```

再来看 `addStorageLocations()` 方法。`addStorageLocations()` 方法会遍历 `DataStorage` 保存的所有 `StorageDirectory` 对象，然后调用 `StorageDirectory.analyseStorage()` 方法分析每个存储目录的状态，对于处于非正常状态的存储目录调用 `StorageDirectory.doRecover()` 方法进行恢复（请参考 `Storage.StorageDirectory` 小节）。当所有的存储目录都处于 `NORMAL` 状态后，调用 `doTransition()` 方法对每个存储目录执行启动参数对应的操作（备份/升级/回滚/恢复/提交）。`addStorageLocations()` 方法的代码如下：

```
private synchronized void addStorageLocations(...) {
    // 1. 确保每个存储目录都处于 NORMAL 状态
    for (Iterator<StorageLocation> it = dataDirs.iterator(); it.hasNext();) {
        File dataDir = it.next().getFile();
        // 对于每个存储目录，构造 StorageDirectory 对象
        StorageDirectory sd = new StorageDirectory(dataDir);
        StorageState curState;
        try {
            // 调用 analyseStorage() 方法分析当前 StorageDirectory 的状态
            curState = sd.analyseStorage(startOpt, this);
            // 根据 curState 恢复状态
            switch (curState) {
                case NORMAL: // 存储目录状态正常，不用执行任何操作
                    break;
                case NON_EXISTENT: // 对于不存在的情况，则直接忽略
                    it.remove();
                    continue;
                case NOT_FORMATTED: // 没有格式化时，调用 format() 方法格式化数据目录
                    format(sd, nsInfo, datanode.getDatanodeUuid());
                    break;
                default: // 对于其他情况，则调用 StorageDirectory.doRecover() 恢复到 NORMAL 状态
                    sd.doRecover(curState);
            }
        } catch (IOException ioe) {
            sd.unlock();
            continue;
        }
        if (isInitialize) {
            addStorageDir(sd);
        }
        // 将 StorageDirectory 对象加入 DataStorage 中保存
        addedStorageDirectories.add(sd);
        dataDirStates.add(curState);
    }

    if (dataDirs.size() == 0 || dataDirStates.size() == 0) {
        if (ignoreExistingDirs) {
            return;
        }
        throw new IOException(
```

```

        "All specified directories are not accessible or do not exist.");
    }

    // 2. 存储状态正常后, 在所有的存储目录下调用 doTransition() 执行 Datanode 启动操作
    for (Iterator<StorageDirectory> it = addedStorageDirectories.iterator();
         it.hasNext(); ) {
        StorageDirectory sd = it.next();
        try {
            // 调用 doTransition() 执行启动操作, 启动选项通过 startOpt 参数传递
            doTransition(datanode, sd, nsInfo, startOpt);
            createStorageID(sd);
        } catch (IOException e) {
            if (!isInitialize) {
                sd.unlock();
                it.remove();
                continue;
            }
            unlockAll();
            throw e;
        }
    }

    // 3. 对于每一个成功执行的存储目录, 写入 VERSION 文件
    this.writeAll(addedStorageDirectories);

    // 4. 将所有的 StorageDirectory 加入 DataStorage.storageDirs 变量中保存
    if (!isInitialize) {
        this.storageDirs.addAll(addedStorageDirectories);
    }
}

```

`addStorageLocations()` 方法调用的 `analyseStorage()` 以及 `doRecover()` 方法我们在 `Storage.StorageDirectory` 小节中已经介绍过了。`format()` 方法用于初始化存储目录, 写入 `VERSION` 文件。存储目录状态恢复到 `NORMAL` 之后, 就需要调用 `doTransition()` 方法执行启动时定义的操作。我们在下面的小节中介绍 `doTransition()` 方法的实现。

(2) doTransition()

将当前存储目录恢复为正常状态之后, `addStorageLocations()` 会调用 `doTransition()` 方法执行启动选项定义的操作。`doTransition()` 方法判断如果启动选项是 `ROLLBACK`, 则调用 `doRollback()` 方法进行回滚操作。如果存储目录记录的文件系统布局版本号 (`VERSION` 文件记录) 与内存中的版本号一致, 则 `Datanode` 正常启动; 如果存储目录记录的版本号小于内存中的版本号, 则调用 `doUpgrade()` 方法升级 (注意 `layoutVersion` 为负数)。`doTransition()` 方法的实现代码如下:

```

private void doTransition( DataNode datanode,
                          StorageDirectory sd,
                          NamespaceInfo nsInfo,

```

Hadoop 2.X HDFS 源码剖析

```
        StartupOption startOpt
        ) throws IOException {
// 如果启动选项是 ROLLBACK, 则调用 doRollback() 进行回滚操作
if (startOpt == StartupOption.ROLLBACK) {
    doRollback(sd, nsInfo);
}
// 检查升级是否成功
readProperties(sd);
checkVersionUpgradable(this.layoutVersion);
assert this.layoutVersion >= HdfsConstants.DATANODE_LAYOUT_VERSION :
    "Future version is not allowed";

boolean federationSupported =
    DataNodeLayoutVersion.supports(
        LayoutVersion.Feature.FEDERATION, layoutVersion);
// 不支持 Federation 部署时, 验证 namespaceID 是否匹配, 不匹配则抛出异常
if (!federationSupported &&
    getNamespaceID() != nsInfo.getNamespaceID()) {
    throw new IOException("Incompatible namespaceIDs in "
        + sd.getRoot().getCanonicalPath() + ": namenode namespaceID = "
        + nsInfo.getNamespaceID() + "; datanode namespaceID = "
        + getNamespaceID());
}

// 支持 Federation 部署时, 验证 clusterID 是否匹配, 不匹配则抛出异常
if (federationSupported
    && !getClusterID().equals(nsInfo.getClusterID())) {
    throw new IOException("Incompatible clusterIDs in "
        + sd.getRoot().getCanonicalPath() + ": namenode clusterID = "
        + nsInfo.getClusterID() + "; datanode clusterID = " + getClusterID());
}

// 磁盘版本号与代码版本号相同, 则 Datanode 正常启动
if (this.layoutVersion == HdfsConstants.DATANODE_LAYOUT_VERSION)
    return;

// 磁盘版本号小于代码版本号, 则调用 doUpgrade() 升级
if (this.layoutVersion > HdfsConstants.DATANODE_LAYOUT_VERSION) {
    doUpgrade(datanode, sd, nsInfo); // upgrade
    return;
}

// 磁盘版本号大于 Datanode 支持的版本号, 则抛出异常
throw new IOException("BUG: The stored LV = " + this.getLayoutVersion()
    + " is newer than the supported LV = "
    + HdfsConstants.DATANODE_LAYOUT_VERSION);
}
```

我们注意到 `doTransition()` 方法会分析启动选项以及存储目录中的文件系统布局版本号，然后分别调用 `doRollback()`、`doUpgrade()` 进行回滚或者升级操作，但是 `doTransition()` 方法中并没有执行提交操作（`finalize`）的入口。这是因为提交操作是 Namenode 通过心跳响应携带的名字节点指令触发 `doFinalize()` 方法执行的，并不是通过 Datanode 启动选项触发的，这里读者需要特别注意。

对于 `doRollback()`、`doUpgrade()` 以及 `doFinalize()` 这三个 `DataStorage` 中与升级相关的方法，我们在下面的小节中进行介绍。

（3）升级相关方法

`DataStorage` 中定义的 `doUpgrade()`、`doRollback()` 以及 `doFinalize()` 这三个方法分别用于升级、回滚以及提交操作。本节我们依次介绍这三个方法的实现，它们的操作逻辑如图 4-16 所示。

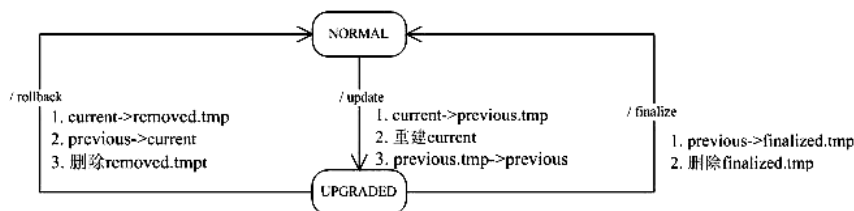


图 4-16 存储目录的升级逻辑

doUpgrade()

`doUpgrade()` 方法用于对单个存储目录下的指定块池目录进行升级，这个存储目录由 `doUpgrade()` 方法的参数 `sd`（`StorageDirectory` 类型）指定，而块池信息则由 `nsInfo` 参数（`NamespaceInfo` 类型）指定。

`doUpgrade()` 方法首先会判断当前存储目录的文件系统布局版本号是否支持 Federation 部署，如果支持则直接更新存储目录的 `VERSION` 文件，然后返回，也就是将块池目录的升级操作委托给 `BlockPoolSliceStorage.doUpgrade()` 方法负责（`recoverTransitionRead()` 方法会调用 `BlockPoolSliceStorage.recoverTransitionRead()` 触发 `BlockPoolSliceStorage.doUpgrade()` 方法）。如果原有版本号不支持 Federation 部署，也就是说，原有的存储目录中并不存在块池目录。则先确保存储目录处于正常状态，并且确认存储目录的 `current` 文件夹存在，然后再进行升级操作。整个升级操作分为 5 个步骤进行。

- 删除存储目录的 `previous` 文件夹，并确认 `previous.tmp` 文件夹不存在。
- 将存储目录的 `current` 目录重命名为 `previous.tmp`。
- 创建 `BlockPoolSliceStorage` 对象，在当前存储目录下创建块池目录，然后格式化（写入 `VERSION` 文件）。接下来创建块池目录的 `current` 文件夹，并与存储目录的 `previous.tmp` 文件夹建立硬链接。
- 在存储目录中写入新的 `VERSION` 文件。
- 最后将存储目录的 `previous.tmp` 文件夹改名为 `previous`，完成整个升级操作。

doUpgrade()方法的代码如下:

```
void doUpgrade(DataNode datanode, StorageDirectory sd, NamespaceInfo nsInfo)
    throws IOException {
    // 如果当前存储目录的文件系统布局版本号支持 Federation 部署
    if (DataNodeLayoutVersion.supports(
        LayoutVersion.Feature.FEDERATION, layoutVersion)) {
        // 直接更新 layoutVersion, 并写入存储目录的 VERSION 文件中
        // 其他升级操作由 BlockPoolSliceStorage.doUpgrade() 方法负责
        layoutVersion = HdfsConstants.DATANODE_LAYOUT_VERSION;
        writeProperties(sd);
        return;
    }

    File curDir = sd.getCurrentDir();
    File prevDir = sd.getPreviousDir();
    File bbwDir = new File(sd.getRoot(), Storage.STORAGE_1_BBW);
    // 确认 current 目录存在
    assert curDir.exists() : "Data node current directory must exist.";
    cleanupDetachDir(new File(curDir, STORAGE_DIR_DETACHED));

    // 1. 删除 previous 目录
    if (prevDir.exists())
        deleteDir(prevDir);
    File tmpDir = sd.getPreviousTmp();
    // 确认 previous.tmp 目录不存在
    assert !tmpDir.exists() :
        "Data node previous.tmp directory must not exist.";

    // 2. 将 current 目录重命名为 previous.tmp
    rename(curDir, tmpDir);

    // 3. 格式化块池, 创建块池目录, 将块池目录的 current 文件夹与存储目录的 previous.tmp 文件夹建立
    硬链接
    File curBpDir = BlockPoolSliceStorage.getBpRoot(nsInfo.getBlockPoolID(), curDir);
    BlockPoolSliceStorage bpStorage=new BlockPoolSliceStorage(nsInfo.getNamespaceID(),
        nsInfo.getBlockPoolID(), nsInfo.getCTime(), nsInfo.getClusterID());
    bpStorage.format(curDir, nsInfo);
    linkAllBlocks(datanode, tmpDir, bbwDir, new File(curBpDir,
        STORAGE_DIR_CURRENT));

    // 4. 在存储目录中写入新的 VERSION 文件
    layoutVersion = HdfsConstants.DATANODE_LAYOUT_VERSION;
    clusterID = nsInfo.getClusterID();
    writeProperties(sd);

    // 5. 将 previous.tmp 目录改名为 previous
    rename(tmpDir, prevDir);
```



```
LOG.info("Upgrade of " + sd.getRoot()+ " is complete");
// 将 BlockPoolSliceStorage 添加到 DataStorage 中保存
addBlockPoolStorage(nsInfo.getBlockPoolID(), bpStorage);
}
```

从代码分析中可以看到，doUpgrade()方法的执行分为两种情况。

- 一种情况是当前 Datanode 存储的文件系统布局版本号支持 Federation 部署。也就是说，当前 Datanode 存储目录中已经保存了若干块池目录，那么当前的升级操作只需要在存储目录的 VERSION 文件中更新 layoutVersion 字段即可，接下来的操作都是在块池目录中进行的，需要由 BlockPoolSliceStorage.doUpgrade()方法执行。
- 另一种情况是当前 Datanode 存储的文件系统版本号比较低，并不支持 Federation 部署。也就是说，当前存储目录中并不存在块池目录，存储目录中只保存了当前唯一的命名空间的所有数据块。在这种情况下，doUpgrade()方法就需要先在当前存储目录中创建并格式化（写入 VERSION 文件）一个新的块池目录，使存储目录的结构符合 Federation 目录结构（请参考 Datanode 磁盘存储结构小节），然后创建 BlockPoolSliceStorage 对象管理这个块池目录，最后 doUpgrade()方法才可以执行升级操作。这里还需要注意一点，升级前的数据块文件都是保存在存储目录的 current 文件夹中的（current 文件夹在升级操作中被重命名为 previous.tmp），这些数据块并不属于任何块池。而升级之后这些数据块要保存在块池目录的 current 文件夹中，也就是原有的数据块都将属于新建的块池，所以 doUpgrade()方法需要建立存储目录的 previous.tmp 文件夹与块池目录 current 文件夹的硬链接。

doRollback()

doRollback()方法用于对单个存储目录下的指定块池目录进行回滚操作，这个存储目录由 doUpgrade()方法的参数 sd（StorageDirectory 类型）指定，而块池信息则由 nsInfo 参数（NamespaceInfo 类型）指定。

doRollback()方法的流程与 doUpgrade()非常类似。doRollback()方法首先判断当前回滚操作是不是在 Federation 部署条件下进行的，如果是则更新存储目录的 VERSION 文件，然后返回，也就是将块池目录的回滚操作委托给 BlockPoolSliceStorage.doRollback()方法负责。如果不是，doRollback()方法会检查存储目录的 previous 目录是否存在，检查是否可以执行回滚操作。完成了上述判断操作后，doRollback()方法才可以进行回滚操作。整个回滚操作分为如下几个步骤进行。

- 首先将存储目录的 current 目录重命名为 removed.tmp。
- 然后将 previous 目录重命名为 current。
- 最后删除 removed.tmp 目录，完成回滚操作。

doRollback()方法的代码如下：

```
void doRollback( StorageDirectory sd,
                NamespaceInfo nsInfo
                ) throws IOException {
```

```
File prevDir = sd.getPreviousDir();
if (!prevDir.exists()) {
    // 回滚操作在 Federation 部署条件下进行
    if (DataNodeLayoutVersion.supports(LayoutVersion.Feature.FEDERATION,
        HdfsConstants.DATANODE_LAYOUT_VERSION)) {
        readProperties(sd, HdfsConstants.DATANODE_LAYOUT_VERSION);
        // 回滚 VERSION 文件
        writeProperties(sd);
    }
    // 直接返回, 块池目录的回滚操作由 BlockPoolSliceStorage.doRollback() 方法负责
    return;
}
DataStorage prevInfo = new DataStorage();
prevInfo.readPreviousVersionProperties(sd);

// 回滚前首先确认当前版本是比回滚版本要新的, 否则抛出异常
if (!(prevInfo.getLayoutVersion() >= HdfsConstants.DATANODE_LAYOUT_VERSION
    && prevInfo.getCTime() <= nsInfo.getCTime()))
    throw new InconsistentFSStateException(sd.getRoot(),
        "Cannot rollback to a newer state.\nDatanode previous state: LV = "
        + prevInfo.getLayoutVersion() + " CTime = " + prevInfo.getCTime()
        + " is newer than the namespace state: LV = "
        + HdfsConstants.DATANODE_LAYOUT_VERSION + " CTime = "
        + nsInfo.getCTime());
// 确认 removed.tmp 不存在
File tmpDir = sd.getRemovedTmp();
assert !tmpDir.exists() : "removed.tmp directory must not exist.";
// 先将 current 目录改名为 removed.tmp
File curDir = sd.getCurrentDir();
assert curDir.exists() : "Current directory must exist.";
rename(curDir, tmpDir);
// 将 previous 目录重命名为 current
rename(prevDir, curDir);
// 删除 removed.tmp, 完成回滚操作
deleteDir(tmpDir);
}
```

doFinalize()

doFinalize()方法用于执行升级提交操作。需要注意的是, doFinalize()方法的入口并不是 doTransition(), 而是 Namenode 通过心跳携带回的名字节点指令。doFinalize()方法的执行逻辑也是比较简单的, 首先将 previous 目录改名为 finalized.tmp, 然后删除 finalized.tmp 即完成了升级。这里的一个小优化是 finalized.tmp 的删除操作是通过单独启动一个新的删除线程执行的, 这是由于 finalized.tmp 文件夹中可能包含比较多的数据块, 删除操作会很慢而导致 doFinalize()方法阻塞, 所以这里单独启动了一个删除线程执行 finalized.tmp 的删除操作。

```
void doFinalize(StorageDirectory sd) throws IOException {
    File prevDir = sd.getPreviousDir();
```

```

if (!prevDir.exists())
    return; // 如果 previous 目录不存在，则没有必要执行提交操作，直接退出执行

final String dataDirPath = sd.getRoot().getCanonicalPath();
assert sd.getCurrentDir().exists() : "Current directory must exist.";
final File tmpDir = sd.getFinalizedTmp(); // finalized.tmp directory
final File bbwDir = new File(sd.getRoot(), Storage.STORAGE_1_BBW);
// 1. 将 previous 目录改名为 finalized.tmp
rename(prevDir, tmpDir);

// 2. 启动一个新的线程，并发删除 finalized.tmp 文件夹
new Daemon(new Runnable() {
    @Override
    public void run() {
        try {
            deleteDir(tmpDir);
            if (bbwDir.exists()) {
                deleteDir(bbwDir);
            }
        } catch (IOException ex) {
            LOG.error("Finalize upgrade for " + dataDirPath + " failed", ex);
        }
        LOG.info("Finalize upgrade for " + dataDirPath + " is complete");
    }
    @Override
    public String toString() { return "Finalize " + dataDirPath; }
}).start();
}

```

7. BlockPoolSliceStorage

在 HDFS Federation 架构中，一个 Datanode 可以保存多个块池的数据块，每个块池的数据块都会分布在 Datanode 所有的存储目录下。HDFS 定义了 BlockPoolSliceStorage 类管理 Datanode 上单个块池的存储空间，DataStorage 类则定义了 bpStorageMap 字段保存 Datanode 上所有块池的 BlockPoolSliceStorage 对象的引用。

DataStorage 类管理着整个 Datanode 的存储，包括 Datanode 定义的多个存储目录。BlockPoolSliceStorage 类则管理着一个块池的存储，包括分布在 Datanode 的多个存储目录下的块池目录（请参考 Datanode 磁盘存储结构小节）。所以 BlockPoolSliceStorage 类的功能与 DataStorage 类基本类似，包括以下几点。

- 格式化一个新的块池存储空间。
- 恢复块池的异常状态。
- 在升级过程中保存上一版本的快照。
- 回滚到上一版本。
- 提交升级，并删除上一版本的快照。

BlockPoolSliceStorage 的入口方法也是 recoverTransitionRead(), 是由上一节中介绍的 DataStorage.recoverTransitionRead()方法调用的, DataStorage.recoverTransitionRead()会首先执行 Datanode 存储的初始化操作, 然后调用 BlockPoolSliceStorage.recoverTransitionRead()执行块池存储的初始化操作。块池存储的升级、回滚以及提交操作是由 BlockPoolSliceStorage 的 doUpgrade()、doRollback()以及 doFinalize()方法执行的, 这三个方法都是在块池目录下执行对应的操作。recoverTransitionRead()、doUpgrade()、doRollback()以及 doFinalize()方法的实现逻辑与 DataStorage 的同名方法基本相同, 这里就不再详细解释了, 请读者自行参考代码。

4.3 文件系统数据集

上一节我们介绍了 DataStorage 类的实现, 这一节介绍 Datanode 最重要的功能——管理与操作数据块功能的实现(创建数据块文件、维护数据块文件与数据块校验文件的对应关系等), 这个功能是由 FsDatasetImpl 类实现的。

Datanode 可以配置多个存储目录保存数据块文件(这里要注意, Datanode 的多个存储目录存储的数据块并不相同, 并且不同的存储目录可以是异构的, 这样的设计可以提高数据块 IO 的吞吐率), 所以 Datanode 将底层数据块的管理抽象为多个层次, 并定义不同的类来实现各个层次上数据块的管理。这些类的定义与引用关系请参考图 4-17, 我们首先介绍这些类的作用, 然后再一一介绍这些类的实现。

- **BlockPoolSlice**: 管理一个指定块池在一个指定存储目录下的所有数据块。由于 Datanode 可以定义多个存储目录, 所以块池的数据块会分布在多个存储目录下。一个块池会拥有多个 BlockPoolSlice 对象, 这个块池对应的所有 BlockPoolSlice 对象共同管理块池的所有数据块。
- **FsVolumeImpl**: 管理 Datanode 一个存储目录下的所有数据块。由于一个存储目录可以存储多个块池的数据块, 所以 FsVolumeImpl 会持有这个存储目录中保存的所有块池的 BlockPoolSlice 对象。
- **FsVolumeList**: Datanode 可以定义多个存储目录, 每个存储目录下的数据块是使用一个 FsVolumeImpl 对象管理的, 所以 Datanode 定义了 FsVolumeList 类保存 Datanode 上所有的 FsVolumeImpl 对象, FsVolumeList 对 FsDatasetImpl 提供类似磁盘的服务。

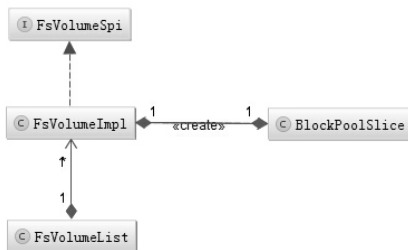


图 4-17 FsDatasetImpl 引用的类

4.3.1 Datanode 上数据块副本的状态

在介绍 BlockPoolSlice、FsVolumeImpl 以及 FsVolumeList 之前，我们先介绍一下 Datanode 上数据块副本的状态。FsDatasetImpl 会持有一个 ReplicaMap 对象，维护 Datanode 上所有数据块副本的状态。Datanode 上保存的数据块副本有下面 5 种状态。

- **FINALIZED**: Datanode 上已经完成写操作的副本。Datanode 定义了 FinalizedReplica 类来保存 FINALIZED 状态副本的信息。
- **RBW (Replica Being Written)**: 刚刚由 HDFS 客户端创建的副本或者进行追加写（append）操作的副本，副本的数据正在被写入，部分数据对于客户端是可见的。Datanode 定义了 ReplicaBeingWritten 类来保存 RBW 状态副本的信息。
- **RUR (Relica Under Recovery)**: 进行块恢复时的副本。Datanode 定义了 ReplicaUnderRecovery 类来保存 RUR 状态副本的信息。
- **RWR (ReplicaWaitingToBeRecovered)**: 如果 Datanode 重启或者宕机，所有 RBW 状态的副本在 Datanode 重启后都将被加载为 RWR 状态，RWR 会等待块恢复操作。Datanode 定义了 ReplicaWaitingToBeRecovered 类来保存 RWR 状态副本的信息。
- **TEMPORARY**: Datanode 之间复制数据块，或者进行集群数据块平衡操作（cluster balance）时，正在写入副本的状态就是 TEMPORARY 状态。和 RBW 不同的是，TEMPORARY 状态的副本对于客户端是不可见的，同时 Datanode 重启时将会直接删除处于 TEMPORARY 状态的副本。Datanode 定义了 ReplicaInPipeline 类来保存 TEMPORARY 状态副本的信息。

这里要注意的是，Datanode 会将不同状态的副本存储到磁盘的不同目录下，也就是说，Datanode 上副本的状态是会被持久化的。HDFS 使用 BlockPoolSlice、FsVolumeImpl 以及 FsVolumeList 类对这些存储目录进行管理，下面我们依次介绍这几个类的实现。

4.3.2 BlockPoolSlice 实现

BlockPoolSlice 类负责管理指定存储目录下指定块池的所有数据块。在 DataStorage 小节中我们介绍过，块池在每个存储目录下都会有一个块池目录存储数据块，换句话说，BlockPollSlice 管理的就是这个块池目录的所有数据块。

在了解 BlockPoolSlice 的实现之前，我们首先介绍块池目录的存储结构。如图 4-18 所示，一个块池目录存在 current/finalized/rbw/lazyPersisit/tmp 等目录。current 目录包含 finalized、rbw 以及 lazyPersisit 三个子目录。finalized 目录保存了所有 FINALIZED 状态的副本，rbw 目录保存了 RBW（正在写）、RWR（等待恢复）、RUR（恢复中）状态的副本，tmp 目录保存了 TEMPORARY 状态的副本。当在客户端发起写请求而创建一个新的副本时，这个副本将会被放到 rbw 目录中。当在数据块备份和集群平衡存储过程中创建一个新的副本时，这个副本就会放到 tmp 目录中。一旦一个副本完成写操作并被提交，它就会被移动到 finalized 目录中。当 Datanode 重启时，tmp 目录中的所有副本将会被删除，rbw 目录中的副本将会被加载为 RWR 状态，finalized 目录中的副本将会被加载为 FINALIZED 状态。lazyPersist 目录为 HDFS 2.6

版本新加入的，用来支持在内存中写入临时数据块副本以及慢持久化（lazy persist）数据块副本的特性，`lazyPersist` 目录就是用于保存这些慢持久化的数据块副本的。

```
data/dfs/data/
├── current
│   └── BP-1079595417-192.168.2.45-1412613236271
│       ├── current
│       │   └── VERSION
│       ├── finalized
│       │   ├── subdir0
│       │   ├── subdir1
│       │   ├── blk_1073741825
│       │   └── blk_1073741825_1001.meta
│       ├── lazyPersist
│       └── rbw
│   ├── dncp_block_verification.log.curr
│   ├── dncp_block_verification.log.prev
│   ├── tmp
│   └── VERSION
└── in_use.lock
```

图 4-18 块池目录结构

需要注意的是，`finalized` 子目录的结构比较特殊，它即包含目录也包含文件。`finalized` 子目录存储了两种类型的文件。

- 数据块文件，如图 4-18 所示，这里的 `blk_1073741825` 就是数据块文件，`1073741825` 是数据块 id。
- `meta` 后缀的校验文件，用来保存数据块文件的校验信息，`blk_1073741825_1001.meta` 就是校验文件，文件名中的 `1001` 是数据块的版本号。

当集群数据量达到一定程度时，`finalized` 目录下的数据块将会非常多。为了便于组织管理，数据块将按照数据块 id 进行散列，拥有相同散列值的数据块将处于同一个子目录下。`finalized` 目录下最多拥有 256 个一级子目录，每个一级子目录下又可以拥有 256 个二级子目录。这种存储组织方式是在 Hadoop 2.6 版本中更新的。在早先版本中，当存储目录中的数据块超过 64 个时（由 `dfs.datanode.numblocks` 配置项配置），将创建 64 个子目录来存储数据块。一个父目录最多可以创建 64 个子目录，一个子目录最多可以存放 64 个数据块（还有 64 个校验文件）以及 64 个子目录。Hadoop 2.6 版本的更新，简化了代码的复杂度，同时又可以直接通过数据块 id 获取数据块存储的文件路径，这种设计方式值得我们积累与学习。

1. BlockPoolSlice 的字段

如图 4-19 所示，`BlockPoolSlice` 类中定义了如下几个比较重要的字段。

- `bpid`：记录当前 `BlockPoolSlice` 对应的块池 id。
- 目录相关字段：保存当前块池目录的 `current` 子目录（`currentDir` 字段）、`finalized` 子目录（`finalizedDir` 字段）、`lazyPersist` 子目录（`lazypersistDir` 字段）、`rbw` 子目录（`rbwDir` 字段）、`tmp` 子目录（`tmpDir` 字段）的引用。

- **dfsUsage**: DU 类型, 描述当前块池目录的磁盘使用情况。

BlockPoolSlice	
LOG	Log
bpid	String
volume	FsVolumeImpl
currentDir	File
finalizedDir	File
lazyPersistDir	File
rbwDir	File
tmpDir	File
DU_CACHE_FILE	String
dfsUsedSaved	boolean
SHUTDOWN_HOOK_PRIORITY	int
deleteDuplicateReplicas	boolean
dfsUsage	DU
getDirectory()	File
getFinalizedDir()	File
getLazyPersistDir()	File
getRbwDir()	File
decDfsUsed(long)	void
getDfsUsed()	long
incDfsUsed(long)	void
loadDfsUsed()	long
saveDfsUsed()	void
createTmpFile(Block)	File
createRbwFile(Block)	File
addBlock(Block, File)	File
activateSavedReplica(Block, File, File)	File
checkDirs()	void
getVolumeMap(ReplicaMap, RamDiskReplicaTracker)	void
recoverTempUnlinkedBlock(File)	File
moveLazyPersistReplicasToFinalized(File)	int
addToReplicasMap(ReplicaMap, File, RamDiskReplicaTracker, boolean)	void
resolveDuplicateReplicas(ReplicaInfo, ReplicaInfo, ReplicaMap)	ReplicaInfo
validateIntegrityAndSetLength(File, long)	long
toString()	String
shutdown()	void

图 4-19 BlockPoolSlice 类结构

2. BlockPoolSlice 的方法

我们知道 FsVolumeImpl 会持有存储目录下所有块池的 BlockPoolSlice 对象, FsVolumeImpl 中多数方法的实现也依赖于底层 BlockPoolSlice 类定义的方法, 所以本节我们学习 BlockPoolSlice 中方法的实现。BlockPoolSlice 定义的方法如图 4-19 所示, 我们将这些方法分为如下几类。

(1) 构造方法

BlockPoolSlice 的构造方法首先会创建 current 目录、finalized 目录、lazyPersist 目录、rbw 目录以及 tmp 目录, 并将这些目录的引用赋值到对应的字段上, 这些目录的结构可以参考图 4-18。完成子目录的创建之后, 构造方法会初始化 dfsUsage 字段, 用来统计块池目录的磁盘使用情况。最后构造方法会添加一个钩子, 当 Datanode 进程结束时保存整个文件系统的磁盘使用信息。

(2) 磁盘空间相关方法

BlockPoolSlice 定义了 5 个用于操作当前块池目录占用磁盘空间的方法, 包括 incDfsUsed()、decDfsUsed()、getDfsUsed()、loadDfsUsed()、saveDfsUsed()。这 5 个方法底层

都是通过调用 `dfsUsage` 字段的对应方法实现的。`dfsUsage` 字段是 `DU` 类型的，`DU` 类是用于统计磁盘使用情况的工具类，这里就不再详细说明了。我们以 `BlockPoolSlice.getDfsUsed()` 方法为例，`getDfsUsed()` 调用了 `dfsUsage.getUsed()` 方法获取块池目录的磁盘使用情况，其他 4 个方法与 `getDfsUsed()` 方法的实现类似，都是调用 `dfsUsage` 对应的方法，`getDfsUsed()` 方法的代码如下：

```
long getDfsUsed() throws IOException {
    return dfsUsage.getUsed();
}
```

（3）块池目录子文件夹方法

我们知道块池目录下会包含若干子文件夹，包括 `current`、`finalized`、`rbw` 以及 `lazyPersist` 等。`BlockPoolSlice` 定义了 4 个用于获取这些子文件夹引用的方法，包括 `getDirectory()`、`getFinalizedDir()`、`getLazyPersistDir()`、`getRbwDir()`。这 4 个方法的实现非常简单，直接返回由构造方法初始化的对应字段即可。代码如下：

```
File getDirectory() {
    return currentDir.getParentFile();
}

File getFinalizedDir() {
    return finalizedDir;
}

File getLazyPersistDir() {
    return lazyPersistDir;
}

File getRbwDir() {
    return rbwDir;
}
```

（4）数据块副本操作方法

`BlockPoolSlice` 类提供了若干在块池目录下操作数据块副本的方法，包括创建不同状态的数据块副本、获取当前块池目录存储副本的状态等，`FsVolumeImpl` 类会通过它持有的 `BlockPoolSlice` 对象调用这些方法完成操作数据块副本的功能。

在块池目录中创建数据块副本

在当前块池目录中创建副本包括以下几种情况：创建 `TEMPORARY` 状态、`RBW` 状态以及 `FINALIZED` 状态的副本等。创建 `TEMPORARY` 状态以及 `RBW` 状态的副本操作都比较简单，直接在 `tmp` 目录以及 `rbw` 目录下通过 `IO` 操作创建数据块副本文件即可。

```
File createTmpFile(Block b) throws IOException {
    File f = new File(tmpDir, b.getBlockName()); // 在 tmp 子目录下创建副本文件
    return DatanodeUtil.createTmpFile(b, f);
}
```



```
File createRbwFile(Block b) throws IOException {
    File f = new File(rbwDir, b.getBlockName()); // 在 rbw 子目录下创建副本文件
    return DatanodeUtil.createTmpFile(b, f);
}
```

创建 FINALIZED 状态的副本则由 `addBlock()` 方法实现，`addBlock()` 方法首先调用 `DatanodeUtil.idToBlockDir()` 方法获得该副本在 `finalized` 目录的存储路径，然后在这个路径下创建数据块副本文件以及校验文件，最后更改 `dfsUsage` 更新磁盘使用情况。

```
File addBlock(Block b, File f) throws IOException {
    // 调用 DatanodeUtil.idToBlockDir() 方法获取副本在 finalized 目录的存储路径
    File blockDir = DatanodeUtil.idToBlockDir(finalizedDir, b.getBlockId());
    if (!blockDir.exists()) {
        if (!blockDir.mkdirs()) {
            // 无法构造存储路径，则抛出异常
            throw new IOException("Failed to mkdirs " + blockDir);
        }
    }
    // 在存储路径中创建数据块副本文件以及校验文件
    File blockFile = FsDatasetImpl.moveBlockFiles(b, f, blockDir);
    File metaFile = FsDatasetUtil.getMetaFile(blockFile, b.getGenerationStamp());
    // 更新 dfsUsage
    dfsUsage.incDfsUsed(b.getNumBytes()+metaFile.length());
    return blockFile;
}
```

我们知道 `finalized` 目录下的副本文件是以多级子目录方式存放的，`DatanodeUtil.idToBlockDir()` 方法提供了确定这个副本存储路径的功能。如下代码所示，`DatanodeUtil.idToBlockDir()` 方法使用 `blockId` 的第三个字节确定一级目录，第二个字节确定二级目录，然后构造存储数据块副本的路径并返回。

```
public static File idToBlockDir(File root, long blockId) {
    // blockId 的第三个字节作为一级目录索引
    int d1 = (int)((blockId >> 16) & 0xff);
    // blockId 的第二个字节作为二级目录索引
    int d2 = (int)((blockId >> 8) & 0xff);
    String path = DataStorage.BLOCK_SUBDIR_PREFIX + d1 + SEP +
        DataStorage.BLOCK_SUBDIR_PREFIX + d2;
    // 构造存储路径并返回
    return new File(root, path);
}
```

获取块池目录存储副本的状态

`Datanode` 在启动时会调用 `BlockPoolSlice.getVolumeMap()` 方法获取当前 `BlockPoolSlice` 持有的所有数据块副本的状态，`DataNode` 类会使用一个 `ReplicaMap`（请参考 `ReplicaMap` 小节）对象保存这些数据块副本的信息以及状态。这里要注意，`ReplicaMap` 对象只保存 FINALIZED

状态以及 RBW 状态的数据块副本。`getVolumeMap()`的代码如下所示,它会首先恢复 `lazyPersist` 状态的副本,将它们转变为 `FINALIZED` 状态,然后 `getVolumeMap()` 方法会调用 `addToReplicasMap()`方法处理 `finalized` 目录以及 `rbw` 目录中保存的所有数据块副本。

```
void getVolumeMap(ReplicaMap volumeMap,
                  final RamDiskReplicaTracker lazyWriteReplicaMap)
    throws IOException {
    // 恢复 lazyPersist 状态的副本,先将它们转变为 FINALIZED 状态
    if (lazyPersistDir.exists()) {
        int numRecovered = moveLazyPersistReplicasToFinalized(lazyPersistDir);
    }

    // 将所有 FINALIZED 状态的副本加入 ReplicaMap 中
    addToReplicasMap(volumeMap, finalizedDir, lazyWriteReplicaMap, true);
    // 将所有 RBW 状态的副本加入 ReplicaMap 中
    addToReplicasMap(volumeMap, rbwDir, lazyWriteReplicaMap, false);
}
```

从上述代码中可以看到,对于 `finalized` 目录以及 `rbw` 目录中保存的数据块副本,`getVolumeMap()`方法调用了 `addToReplicasMap()`方法进行处理,只不过 `isFinalized` 参数一个是 `true`,一个是 `false`。`addToReplicasMap()`方法会遍历 `dir` 参数指定的文件夹下的所有副本文件,如果是 `finalized` 文件夹中的,则加载为 `FINALIZED` 状态;如果不是 `finalized` 文件夹中的,则判断当前文件夹中是否存在 `.restart` 文件(`Datanode` 快速重启时会在磁盘上保存一个文件,里面存储了一个大概的重启时间,用于恢复文件的 IO 流),如果存在这个文件,并且时间还在重启窗口内,则当前数据块副本可以恢复为 `RBW` 状态,将当前数据块加载为 `RBW` 状态;如果不存在这个文件,则将当前数据块加载为 `RWR` 状态。

```
void addToReplicasMap(ReplicaMap volumeMap, File dir,
                     final RamDiskReplicaTracker lazyWriteReplicaMap,
                     boolean isFinalized)
    throws IOException {
    // 循环遍历文件夹中的所有文件
    File files[] = FileUtil.listFiles(dir);
    for (File file : files) {
        if (file.isDirectory()) {
            // 如果是文件夹,则递归调用 addToReplicasMap() 方法
            addToReplicasMap(volumeMap, file, lazyWriteReplicaMap, isFinalized);
        }

        // 如果文件夹中存在临时文件,则首先进行恢复操作
        if (isFinalized && FsDatasetUtil.isUnlinkTmpFile(file)) {
            file = recoverTempUnlinkedBlock(file);
            if (file == null) { // 如果临时文件对应的源文件存在,那么跳过该临时文件
                continue;
            }
        }
    }
}
```

```

// 如果是 finalized 文件夹, 则将所有数据块加载为 FINALIZED 状态
if (isFinalized) {
    newReplica = new FinalizedReplica(blockId,
        file.length(), genStamp, volume, file.getParentFile());
} else {
    boolean loadRwr = true;
    File restartMeta = new File(file.getParent() +
        File.pathSeparator + "." + file.getName() + ".restart");
    Scanner sc = null;
    try {
        sc = new Scanner(restartMeta);
        // 重启元数据文件存在, 并且当前时间在重启窗口内
        if (sc.hasNextLong() && (sc.nextLong() > Time.now())) {
            // 将数据块加载为 RBW 状态
            newReplica = new ReplicaBeingWritten(blockId,
                validateIntegrityAndSetLength(file, genStamp),
                genStamp, volume, file.getParentFile(), null, 0);
            loadRwr = false;
        }
        sc.close();
        if (!restartMeta.delete()) {
            FsDatasetImpl.LOG.warn("Failed to delete restart meta file: " +
                restartMeta.getPath());
        }
    } catch (FileNotFoundException fnfe) {}
    finally {
        if (sc != null) {
            sc.close();
        }
    }
    // 重启元数据文件不存在, 将数据块加载为 RWR 状态
    if (loadRwr) {
        newReplica = new ReplicaWaitingToBeRecovered(blockId,
            validateIntegrityAndSetLength(file, genStamp),
            genStamp, volume, file.getParentFile());
    }
}
// 将数据块信息放入 volumeMap 中
ReplicaInfo oldReplica = volumeMap.get(bpid, newReplica.getBlockId());
if (oldReplica == null) {
    volumeMap.add(bpid, newReplica);
} else {
    newReplica = resolveDuplicateReplicas(newReplica, oldReplica, volumeMap);
}
// ...
}
}

```

recoverTempUnlinkedBlock()

`addToReplicasMap()`方法调用了 `recoverTempUnlinkedBlock()`恢复临时文件（*.unlinked 文件）。我们首先看一下临时文件的概念。`Datanode` 在升级过程中或者回滚到上一版本前，`previous` 目录和 `current` 目录会使用硬链接保留相同数据块的引用。如果这时客户端对任意数据块进行追加写操作，将会同时影响两个引用，也就是影响上一版本数据的快照。这时需要去除 `current` 目录对该数据块的硬链接，保证后续对该数据块的修改不会影响 `previous` 目录中保存的上一版本数据块的快照。

`Datanode` 目前的逻辑是，将需要更改的数据块先复制一份，创建一个临时文件并命名为“\${blockName}.unlinked”，这样临时文件将不会存在任何硬链接，且内容与源文件完全相同。当临时文件创建成功后，再把临时文件改名为源文件。这时 `current` 和 `previous` 中将存在两个不同的数据块文件，但内容却是完全相同的。

注意：早先版本使用 `detach` 目录作为消除硬链接的目录。

`recoverTempUnlinkedBlock()`方法就是用来对临时文件（*.unlinked 文件）进行恢复操作的。恢复操作的逻辑是首先判断源文件是否存在，如果不存在则将临时文件改名为源文件；如果源文件存在，则将临时文件直接删除，非常简单。

```
File recoverTempUnlinkedBlock(File unlinkedTmp) throws IOException {
    File blockFile = FsDatasetUtil.getOrigFile(unlinkedTmp);
    if (blockFile.exists()) {
        // 源文件存在，则删除临时文件，如果删除失败则抛出异常
        if (!unlinkedTmp.delete()) {
            throw new IOException("Unable to cleanup unlinked tmp file " +
                unlinkedTmp);
        }
        return null;
    } else {
        // 源文件不存在，则将临时文件改名为源文件
        if (!unlinkedTmp.renameTo(blockFile)) {
            throw new IOException("Unable to rename unlinked tmp file " +
                unlinkedTmp);
        }
        return blockFile;
    }
}
```

4.3.3 FsVolumeImpl 实现

`Datanode` 可以配置多个存储目录存储数据块文件。每一个存储目录下的数据块管理功能都由一个单独的 `FsVolumeImpl` 对象负责。而每一个存储目录又可以保存多个块池的数据块，其中每一个块池数据块的管理都由一个 `BlockPoolSlice` 对象负责，因此 `FsVolumeImpl` 对象会持有多个 `BlockPoolSlice` 对象的引用。`BlockPoolSlice` 类的实现我们在上一节中已经介绍过了，

本节介绍 FsVolumeImpl 类的实现。

1. FsVolumeSpi 接口定义

如图 4-20 所示, FsVolumeSpi 是 FsVolumeImpl 的根接口, 下面介绍 FsVolumeSpi 的接口定义。

FsVolumeSpi 作为 FsVolumeImpl 的根接口, 定义了如下几个方法, 如图 4-21 所示。

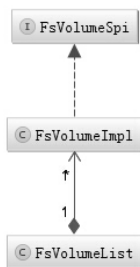


图 4-20 FsVolumeSpi 类图

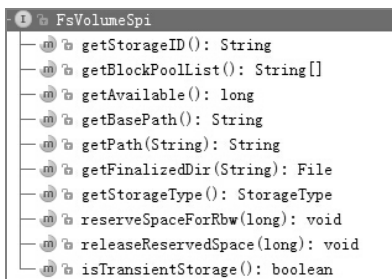


图 4-21 FsVolumeSpi 的方法

- `getStorageID()`: 获取当前存储目录的 `StorageUuid`。
- `getBlockPoolList()`: 获取当前存储目录下的块池列表, 这里返回的是当前存储目录下保存的所有块池的块池 id (`blockPoolId`)。
- `getAvailabe()`: 获取当前存储目录的可用空间。
- `getBasePath()`: 获取当前存储目录的根路径。
- `getPath()`: 获取指定块池在当前存储目录下的路径。
- `getFinalizedDir()`: 获取指定块池在当前存储目录下的 `finalized` 目录。
- `getStorageType()`: 获取当前存储目录的存储类型, 存储目录类型可以是内存、SSD 或者磁盘。
- `reserveSpcaeForRbw()`: 对 RBW 状态的数据块预留磁盘空间, 这样写入数据块时就不会出现磁盘空间不足的情况了。
- `releaseReservedSpace()`: 释放预留的空间。
- `isTransientStorage()`: 判断当前卷是否有与之对应的持久化磁盘。

注意: 这里的 `reserveSpcaeForRbw()`、`releaseReservedSpace()` 以及 `isTransientStorage()` 三个方法是在 Hadoop2.6 版本中新加入的方法。

2. FsVolumelmpl 的字段和方法

FsVolumeImpl 为了实现 FsVolumeSpi 接口, 定义了若干字段, 如图 4-22 所示。

- `dataset`: 这个字段主要用于加锁操作。
- `storageID`: 当前存储目录对应的 `StorageDirectory` 的 `storageID`。
- `currentDir`: 当前存储目录下 `current` 文件夹的引用。
- `usage`: 当前存储目录的磁盘使用情况。

- **reserved**: 当前存储目录的预留磁盘空间大小。
- **bpSlices**: 当前 `FsVolumeImpl` 下所有 `BlockPoolSlice` 的引用, 是 `blockPoolId` -> `BlockPoolSlice` 的映射。
- **cacheExecutor**: 线程池, 用来处理添加到缓存中的新的数据块。
- **reservedForRbw**: 预留的磁盘空间。

FsVolumeImpl	
dataset	FsDatasetImpl
storageID	String
storageType	StorageType
bpSlices	Map<String, BlockPoolSlice>
currentDir	File
usage	DF
reserved	long
reservedForRbw	AtomicLong
configuredCapacity	long
cacheExecutor	ThreadPoolExecutor

图 4-22 FsVolumeImpl 字段

了解了 `FsVolumeImpl` 定义的字段, 我们来学习 `FsVolumeImpl` 定义的方法。 `FsVolumeImpl` 的方法可以分为三类。

- **获取存储目录的存储状态**: 包括 `getDfsUsed()`、`getBlockPoolUsed()`、`getCapacity()`、`getAvailable()`、`getReserved()`等方法。这些方法都用于获取存储目录的磁盘使用状态, 底层基本都是通过调用 `BlockPoolSlice` 对应方法获得返回值的。这里我们以 `getDfsUsed()`方法为例, 其他方法的实现与 `getDfsUsed()`类似。 `getDfsUsed()`方法调用了这个存储目录管理的所有 `BlockPoolSlice` 对象的 `getDfsUsed()`方法, 并将这些值累加并返回。代码如下:

```
long getDfsUsed() throws IOException {
    long dfsUsed = 0;
    synchronized(dataset) {
        for(BlockPoolSlice s : bpSlices.values()) {
            dfsUsed += s.getDfsUsed();
        }
    }
    return dfsUsed;
}
```

- **获取存储目录的子目录**: 包括 `getCurrentDir()`、`getRbwDir()`以及 `getLazyPersistDir()`三个方法, 用于获取存储目录下的特定子目录。其中 `getCurrentDir()`用于获取存储目录下的 `current` 子目录, 方法底层是直接返回了 `FsVolumeImpl.currentDir` 字段; 而 `getRbwDir()`和 `getLazyPersistDir()`两个方法则用于获取指定块池目录下的 `rbw` 文件夹以及 `lazyPersist` 文件夹, 这两个方法是通过调用 `bpId` 对应的 `BlockPoolSlice` 对象的方法实现的。代码如下:

```
File getCurrentDir() {
    return currentDir;
}
```

```
File getRbwDir(String bpid) throws IOException {
    return getBlockPoolSlice(bpid).getRbwDir();
}
```

```
File getLazyPersistDir(String bpid) throws IOException {
    return getBlockPoolSlice(bpid).getLazypersistDir();
}
```

- 数据块操作：包括添加数据块副本、获取数据块副本信息等操作。这些方法底层都是在 `BlockPoolSlice` 对象上调用对应的方法，代码比较简单，不再赘述。
- 块池操作：在当前存储目录下添加或者删除一个块池的存储，大多是对 `bpSlices` 属性进行操作。这里以 `addBlockPool()` 为例，构造一个新的 `BlockPoolSlice` 对象，并放入 `bpSlices` 字段中保存即可。

```
void addBlockPool(String bpid, Configuration conf) throws IOException {
    File bpdire = new File(currentDir, bpid);
    BlockPoolSlice bp = new BlockPoolSlice(bpid, this, bpdire, conf); //构造 BlockPoolSlice
    bpSlices.put(bpid, bp); // 放入 bpSlices 字段中保存
}
```

4.3.4 FsVolumeList 实现

我们知道 `Datanode` 可以设置多个存储目录，每一个存储目录的数据块都是由一个 `FsVolumeImpl` 对象负责管理的。对于 `Datanode` 定义的多个存储目录，HDFS 提供了接口类——`FsVolumeList` 进行统一管理，`FsVolumeList` 使用一个线性表保存所有的 `FsVolumeImpl` 对象，并提供统一的接口在所有的 `FsVolumeImpl` 对象上执行操作。`FsVolumeList` 定义的字段和方法如图 4-23 所示。

FsVolumeList	
① volumes	List<FsVolumeImpl>
② blockChooser	VolumeChoosingPolicy<FsVolumeImpl>
③ numFailedVolumes	int
④ numberOfFailedVolumes()	int
⑤ getNextVolume(StorageType, long)	FsVolumeImpl
⑥ getNextTransientVolume(long)	FsVolumeImpl
⑦ getDfsUsed()	long
⑧ getBlockPoolUsed(String)	long
⑨ getCapacity()	long
⑩ getRemaining()	long
⑪ getAllVolumesMap(String, ReplicaMap, RandDiskReplicaTracker)	void
⑫ checkDirs()	List<FsVolumeImpl>
⑬ toString()	String
⑭ addVolume(FsVolumeImpl)	void
⑮ removeVolume(String)	void
⑯ addBlockPool(String, Configuration)	void
⑰ removeBlockPool(String)	void
⑱ shutdown()	void

图 4-23 FsVolumeList 类结构

FsVolumeList 定义了两个重要的字段。

- **List<FsVolumeImpl> volumes:** 保存 Datanode 配置的所有 FsVolumeImpl 对象。注意, 这个集合是一个只读集合, 不可以进行修改。
- **VolumeChoosingPolicy<FsVolumeImpl>blockChooser:** 选择一个存储目录对应的 FsVolumeImpl 对象来存放数据块副本。目前有两种策略。
 - **AvailableSpaceVolumeChoosingPolicy:** 选择有更多可用空间的存储目录来存放副本。
 - **RoundRobinVolumeChoosingPolicy:** 轮询策略, 轮询直到选择出第一个有足够空间的存储目录来存放副本。

FsVolumeList 主要提供以下三种类型的操作。

- **获取 Datanode 节点状态操作:** FsVolumeList 提供了获取 Datanode 总容量、剩余容量、dfs 使用量和数据块信息 (getVolumeMap()和 getAllVolumesMap()) 等方法。
- **BlockPool 相关:** FsVolumeList 提供了添加和删除 BlockPool 的方法 (addBlockPool()、removeBlockPool()), 用于在所有的 FsVolumeImpl 上添加和删除一个块池存储目录。这里我们学习 addBlockPool()方法, 这个方法遍历所有的 FsVolumeImpl 对象, 对于每一个 FsVolumeImpl 对象都启动一个独立的线程级联调用该对象的 addBlockPool()方法, 在该对象的存储目录下添加新的块池存储目录, 然后等待所有线程执行完毕后返回。

```
void addBlockPool(final String bpid, final Configuration conf) throws IOException {
    final List<IOException> exceptions = Collections.synchronizedList(
        new ArrayList<IOException>());
    List<Thread> blockPoolAddingThreads = new ArrayList<Thread>();
    for (final FsVolumeImpl v : volumes) {
        // 对于每一个 FsVolumeImpl 级联启动一个独立的线程
        Thread t = new Thread() {
            public void run() {
                try {
                    // 调用 FsVolumeImpl.addBlockPool() 在对应存储目录下添加块池目录
                    v.addBlockPool(bpid, conf);
                } catch (IOException ioe) {
                    exceptions.add(ioe);
                }
            }
        };
        blockPoolAddingThreads.add(t);
        t.start();
    }
    // 等待所有线程执行完毕
    for (Thread t : blockPoolAddingThreads) {
        try {
            t.join();
        } catch (InterruptedException ie) {
            throw new IOException(ie);
        }
    }
}
```



```

    }
}
if (!exceptions.isEmpty()) {
    throw exceptions.get(0);
}
}

```

- 获取一个可以存储副本的存储目录：对应于 `getNextVolume()` 方法，这个方法的实现比较简单，就是在 `blockChooser` 字段上调用 `chooseVolume()` 方法。可以有两种策略，即轮询与可用空间优先（请参考上面的字段介绍部分）。Datanode 在存储一个新的数据块副本时，会首先调用这个方法获取一个可以存储这个数据块副本的存储目录，然后在这个存储目录对应的 `FsVolumeImpl` 对象上调用 `createRbwFile()` 或者 `createTemporaryFile()` 方法，创建临时数据块文件进行写操作。

```

synchronized FsVolumeImpl getNextVolume(StorageType storageType,
    long blockSize) throws IOException {
    final List<FsVolumeImpl> list = new ArrayList<FsVolumeImpl>(volumes.size());
    for(FsVolumeImpl v : volumes) {
        if (v.getStorageType() == storageType) {
            list.add(v);
        }
    }
    // 调用 blockChooser.chooseVolume() 方法获取一个可以存储副本的存储目录
    return blockChooser.chooseVolume(list, blockSize);
}

```

下面我们看一下 `chooseVolume()`—`RoundRobinVolumeChoosingPolicy` 的实现，轮询策略会轮询所有 `FsVolumeImpl` 直到找到第一个空间足够的 `FsVolumeImpl` 对象。

```

public synchronized V chooseVolume(final List<V> volumes, long blockSize)
    throws IOException {
    // ...

    while (true) {
        // 返回第一个有足够空间的 volume
        final V volume = volumes.get(curVolume);
        curVolume = (curVolume + 1) % volumes.size();
        long availableVolumeSize = volume.getAvailable();
        if (availableVolumeSize > blockSize) {
            return volume;
        }

        if (availableVolumeSize > maxAvailable) {
            maxAvailable = availableVolumeSize;
        }

        if (curVolume == startVolume) {
            throw new DiskOutOfSpaceException("Out of space: "
                + "The volume with the most available space (" + maxAvailable

```

```

        + " B) is less than the block size (" + blockSize + " B).");
    }
}
}

```

4.3.5 FsDatasetImpl 实现

前面我们依次介绍了 `BlockPoolSlice`、`FsVolumeImpl` 以及 `FsVolumeList` 的实现，`BlockPoolSlice` 负责管理单个存储目录下单个块池的所有数据块；`FsVolumeImpl` 则负责管理一个完整的存储目录下所有的数据块，也就包括了在这个存储目录下多个 `BlockPoolSlice` 对象的引用。而我们知道 `Datanode` 可以定义多个存储目录，也就是定义多个 `FsVolumeImpl` 对象，在 HDFS 中使用 `FsVolumeList` 对象统一管理 `Datanode` 上定义的所有 `FsVolumeImpl` 对象。了解了这三个类的实现，下面我们介绍负责管理 `Datanode` 上所有数据块功能的类——`FsDatasetImpl` 的实现。

如图 4-24 所示，`FsDatasetSpi` 是 `FsDatasetImpl` 的根接口，定义了 `Datanode` 管理数据块的接口方法。`FsDatasetImpl` 会通过持有一个 `FsVolumeList` 对象对 `Datanode` 上定义的所有存储目录下的数据块进行管理操作，`FsDatasetImpl` 还会持有一个 `DataStorage` 对象对 `Datanode` 的存储空间进行操作。同时 `FsDatasetImpl` 会通过持有一个 `ReplicaMap` 对象维护 `Datanode` 上所有副本的信息与状态。

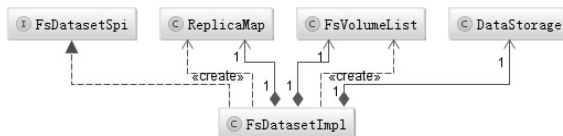


图 4-24 `FsDatasetImpl` 类图

`FsVolumeList` 和 `DataStorage` 两个对象在前面的小节中已经重点介绍过了，本节首先介绍 `ReplicaMap` 和 `FsDatasetSpi` 两个类，然后介绍 `FsDatasetImpl` 的代码实现。

1. ReplicaMap

`ReplicaMap` 保存了 `Datanode` 上所有数据块副本的信息。如下代码所示，`ReplicaMap` 维护一个 `Map`，保存 `Datanode` 上所有块池（`blockPoolId` 标识）与这个块池保存的数据块副本信息的映射关系。数据块副本的信息是通过 `ReplicaInfo` 类描述的。

```

private final Map<String, Map<Long, ReplicaInfo>> map =
    new HashMap<String, Map<Long, ReplicaInfo>>();

```

`ReplicaMap` 的方法大多是操作这个 `Map` 数据结构的，比较简单，我们就不再赘述了。

2. ReplicaInfo

`ReplicaInfo` 是一个抽象类，用来描述 `Datanode` 上保存的数据块副本信息。`ReplicaInfo` 类对外提供了 `getBlockFile()`、`getMetaFile()`、`getVolume()`、`getStorageUuid()` 等方法，分别用于获

取副本的数据块文件、校验文件、副本所在存储目录的 `FsVolumeImpl` 对象以及副本所在存储目录的 `storageUuid` 信息。这里的实现都比较简单，直接返回 `ReplicaInfo` 持有的相应字段就可以了。

如图 4-25 所示，`ReplicaInfo` 有如下几个子类分别用来描述处于不同状态的数据块副本。

- `ReplicaBeingWritten`——对应 RBW 状态的副本，副本正通过数据流管道写入 Datanode。
- `ReplicaUnderRecovery`——对应 RUR 状态的副本，副本正在进行恢复操作。
- `ReplicaWaitingToBeRecovered`——对应 RWR 状态的副本，副本等待恢复操作。
- `FinalizedReplica`——对应 FINALIZED 状态的副本，副本已经完成写操作，并且已经提交。

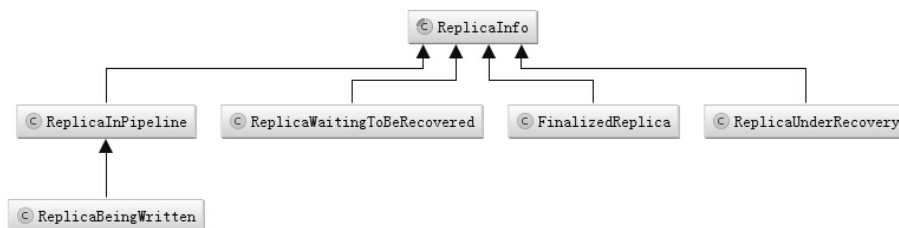


图 4-25 `ReplicaInfo` 继承关系图

这里我们讲解 `ReplicaInfo.unlinkBlock()` 方法，这个方法用于移除数据块上的硬链接。通过将数据块文件移动到一个临时的位置，然后再移动回来即可移除硬链接。为什么要移除硬链接呢？考虑如下情况：`Datanode` 在升级过程中或者回滚到上一版本前，`previous` 目录和 `current` 目录使用硬链接保留了对相同数据块的引用。如果这时客户端对任意数据块进行追加写（append）操作，将会同时影响两个引用，也就影响了上一版本的数据。这时需要去除 `current` 目录对该数据块的硬链接，保证后续对该数据块的修改不会影响 `previous` 目录。

`unlinkBlock()` 方法会调用 `unlinkFile()` 方法去除数据块副本文件以及校验和文件的硬链接，`unlinkFile()` 方法会将需要更改的文件先复制一份，创建一个临时文件并命名为 “`${fileName}.unlinked`”，这样临时文件将不会存在任何硬链接，且内容与源文件完全相同。当临时文件创建成功后，再把临时文件改名为源文件。这时 `current` 和 `previous` 中将存在两个不同的文件，但内容却是完全相同的。临时文件的恢复操作在 `BlockPoolSlice` 小节中已有介绍，读者可以参考一并学习。

```

public boolean unlinkBlock(int numLinks) throws IOException {
    // 如果已经移除了硬链接，则直接返回
    if (isUnlinked()) {
        return false;
    }
    File file = getBlockFile();
    if (file == null || getVolume() == null) {
        throw new IOException("detachBlock:Block not found. " + this);
    }
}

```

Hadoop 2.X HDFS 源码剖析

```
    }
    File meta = getMetaFile();
    // 移除数据块文件硬链接
    if (HardLink.getLinkCount(file) > numLinks) {
        unlinkFile(file, this);
    }
    // 移除校验文件硬链接
    if (HardLink.getLinkCount(meta) > numLinks) {
        unlinkFile(meta, this);
    }
    setUnlinked();
    return true;
}

private void unlinkFile(File file, Block b) throws IOException {
    // 构造临时文件，临时文件在同一目录下，在数据块文件名后加后缀“.unlinked”
    File tmpFile = DatanodeUtil.createTmpFile(b, DatanodeUtil.getUnlinkTmpFile(file));
    try {
        FileInputStream in = new FileInputStream(file);
        try {
            FileOutputStream out = new FileOutputStream(tmpFile);
            try {
                // 将源文件拷贝至临时文件中
                IOUtils.copyBytes(in, out, 16*1024);
            } finally {
                out.close();
            }
        } finally {
            in.close();
        }
    }
    if (file.length() != tmpFile.length()) {
        throw new IOException("Copy of file " + file + " size " + file.length()+
            " into file " + tmpFile +
            " resulted in a size of " + tmpFile.length());
    }
    // 用临时文件替换源文件，然后删除临时文件
    FileUtil.replaceFile(tmpFile, file);
} catch (IOException e) {
    boolean done = tmpFile.delete();
    if (!done) {
        DataNode.LOG.info("detachFile failed to delete temporary file " +
            tmpFile);
    }
    throw e;
}
}
```

3. FsDatasetSpi

如图 4-26 所示, FsDatasetSpi 是 FsDatasetImpl 的根接口, 继承自 FSDatasetMBean 接口, 定义了 Datanode 上管理与操作数据块副本的接口。

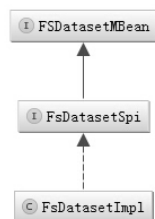


图 4-26 FsDatasetSpi 类图

这里我们重点看一下 FsDatasetSpi 定义了哪些方法。FsDatasetSpi 定义的方法较多, 我们将这些方法分为如下几类。

- 操作底层 FsVolumeList 以及 DataStorage 相关方法: getVolumes()、getVolume()、getStorage()、getVolumeInfoMap()等方法。
- 操作数据块相关方法: getFinalizedBlocks()、getStoredBlock()、getBlockInputStream()、append()、finalizeBlock()等方法。
- 操作缓存相关方法: cache()、uncache()、isCached()等方法。
- 操作块池方法: addBlockPool()、shutdownBlockPool()、deleteBlockPool()等方法。
- 数据块汇报相关方法: getStorageReports()、getBlockReports()、getCacheReport()。
- 回收站相关方法: enableTrash()、restoreTrash()、trashEnabled()。

4. FsDatasetImpl

FsDatasetImpl 实现了 FsDatasetSpi 接口, 本节介绍 FsDatasetImpl 类的实现。

(1) FsDatasetImpl 字段

FsDatasetImpl 定义的字段如图 4-27 所示。

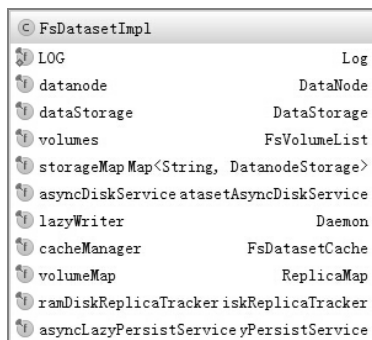


图 4-27 FsDatasetImpl 类字段

- **datanode**: 当前数据节点的 Datanode 对象的引用。
- **dataStorage**: DataStorage 对象的引用, 在 FsDatasetImpl 中调用这个对象开启和关闭存储空间回收站功能。
- **volumes**: FsVolumeList 对象的引用, FsDatasetImpl 使用 FsVolumeList 对象提供的功能管理 Datanode 上的所有数据块, 并且通过对 volumes 字段的修改添加和删除新的存储目录。

- **storageMap**: 在 `FsDatasetImpl` 中, 对于每一个存储目录都会构造一个 `DataStorage` 对象, `storageMap` 字段就是用来保存这些存储的, 是 `storageUuid->DataStorage` 的映射。`FsDatasetImpl` 可以通过 `storageUuid` 获取对应的 `DataStorage` 对象。
- **volumeMap**: `ReplicaMap` 类型, 用于记录 `Datanode` 上所有数据块副本的信息。
- **cacheManager**: `FsDatasetCache` 类型, 用于将数据块缓存到内存中的工具类。
- **asyncDiskService**: `FsDatasetAsyncDiskService` 类型, 用于在 `Datanode` 磁盘存储上进行一些异步的 IO 操作。例如在进行删除数据块副本操作时, 会调用 `FsDatasetAsyncDiskService.deleteAsync()` 异步地将数据块文件从磁盘上删除。

(2) `FsDatasetImpl` 方法

介绍完 `FsDatasetImpl` 定义的字段后, 我们学习 `FsDatasetImpl` 的方法。`FsDatasetImpl` 定义的方法非常多, 我们把这些方法分为如下几类进行介绍。

- 操作底层 `FsVolumeList` 以及 `DataStorage` 相关方法。
- 操作数据块相关方法。
- 操作缓存相关方法。
- 操作块池方法。
- 数据块汇报相关方法。
- 回收站相关方法。

操作底层 `FsVolumeList` 以及 `DataStorage` 相关方法

包括 `getVolumes()`、`getVolume()`、`getStorage()`和 `getVolumeInfoMap()`方法, 分别用于获取 `FsVolumeImpl` 对象的列表、获取指定存储目录对应的 `FsVolumeImpl` 对象、获取指定存储目录对应的 `DataStorage` 对象以及获取存储目录的卷信息。这类方法的实现都比较简单, 基本都是直接返回对应的 `FsVolumeList` 与 `DataStorage` 对象。

比较特别的是 `getVolumeInfoMap()`方法返回的是 `VolumeInfo` 对象的集合。`VolumeInfo` 是 `FsDatasetImpl` 的内部类, 用于描述一个 `FsVolumeImpl` 对应存储目录的存储信息。`VolumeInfo` 的代码如下:

```
private static class VolumeInfo {
    final String directory;    // 存储目录的路径
    final long usedSpace;      // 存储目录中已经使用的空间
    final long freeSpace;      // 存储目录中可用的空间
    final long reservedSpace;  // 存储目录中预留的空间

    VolumeInfo(FsVolumeImpl v, long usedSpace, long freeSpace) {
        this.directory = v.toString();
        this.usedSpace = usedSpace;
        this.freeSpace = freeSpace;
        this.reservedSpace = v.getReserved();
    }
}
```

数据块副本信息相关方法

我们知道 `FsDatasetImpl` 使用 `ReplicaMap` 记录 `Datanode` 存储的所有副本信息, `FsDatasetImpl` 定义了若干获取数据块副本信息的方法, 我们将这部分方法分为三类。

获取数据块副本信息相关方法

这类方法提供了获取数据块副本相关信息的功能, 包括 `getStoredBlock()`、`getFinalizedBlocks()`、`getReplicaString()`、`getLength()`等方法。这些方法的实现都比较简单, 都是从 `FsDatasetImpl.volumeMap` (`ReplicaMap` 类型) 字段中提取出对应的副本信息 (`ReplicaInfo` 对象保存), 然后构造 `Block` 或者 `Replica` 对象返回即可。

这里我们以 `getStoredBlock()`方法为例, `getStoredBlock()`方法用于获取指定数据块的 `Block` 对象。`Block` 类封装了一个数据块副本的 `id`、数据块副本文件的长度以及时间戳。这个方法首先调用 `getFile()`方法从 `FsDatasetImpl.volumeMap` 字段中取出 `ReplicaInfo` 对象, 然后通过 `ReplicaInfo` 对象获取副本的数据块文件引用, 最后构造一个 `Block` 对象并返回。代码如下:

```
public synchronized Block getStoredBlock(String bpid, long blkid)
    throws IOException {
    // 调用 getFile() 获取副本的数据块文件引用
    File blockfile = getFile(bpid, blkid, false);
    if (blockfile == null) {
        return null;
    }
    // 获取副本的校验和文件引用
    final File metafile = FsDatasetUtil.findMetaFile(blockfile);
    // 获取副本的时间戳
    final long gs = FsDatasetUtil.parseGenerationStamp(blockfile, metafile);
    // 构造 Block 对象返回
    return new Block(blkid, blockfile.length(), gs);
}

File getFile(final String bpid, final long blockId, boolean touch) {
    // 通过 volumeMap 获取副本信息
    ReplicaInfo info = volumeMap.get(bpid, blockId);
    if (info != null) {
        if (touch && info.getVolume().isTransientStorage()) {
            ramDiskReplicaTracker.touch(bpid, blockId);
            datanode.getMetrics().incrRamDiskBlocksReadHits();
        }
        // 返回副本信息记录的数据块文件引用
        return info.getBlockFile();
    }
    return null;
}
```

获取数据块副本 IO 流相关方法

包括 `getBlockInputStream()`和 `getMetaDataInputStream()`两个方法, 分别用于获取副本的数

据块文件以及校验文件的 IO 流。这两个方法的实现非常简单，就是通过上面介绍的 `getFile()` 方法获得数据块文件引用，然后打开文件构造 IO 流返回即可。代码比较简单，我们就不再赘述了。

checkAndUpdate()方法

`FsDatasetImpl` 使用 `volumeMap` 字段记录所有副本的信息，`volumeMap` 对象保存在内存中，这就可能出现内存中 `volumeMap` 记录的副本状态与磁盘上存储的副本状态不一致的情况。

`FsDatasetImpl` 提供了 `checkAndUpdate()` 方法同步内存中的副本状态和磁盘上存储的副本状态。这个方法是在 `DirectoryScanner.run()` 中调用的，`DirectoryScanner` 首先调用 `scan()` 方法获取内存与磁盘不一致的数据块副本引用，然后调用 `checkAndUpdate()` 方法同步两者的状态。`checkAndUpdate` 方法的逻辑如下：

- 如果磁盘上没有这个数据块，则从 `volumeMap` 中删除数据块。
- 如果磁盘上存在这个数据块，`volumeMap` 中不存在，则添加到 `volumeMap` 中。
- 如果两者的时间戳不匹配，则用正确的时间戳更新副本信息。
- 如果内存中数据块的长度与磁盘上数据块的长度不匹配，则将数据块标识为损坏 (corrupt)，然后更新副本的长度。
- 如果内存中记录的数据块文件与磁盘上的数据块文件不匹配，则更新副本的数据块文件引用。

对于情况 1，要特别注意从内存中删除数据块副本时，不仅仅要从 `volumeMap` 中删除，还需要调用 `blockScanner.deleteBlock()` 方法从 `BlockScanner` 对象中删除。对于情况 4，当出现数据块长度不一致的情况时，还需要调用 `datanode.reportBadBlocks()` 向 `Namenode` 汇报这个数据块副本已经损坏。对于情况 5，如果 `volumeMap` 中记录的数据块文件与磁盘扫描的数据块文件不一致，则调用 `BlockPoolSlice.resolveDuplicateReplicas()` 方法判断保留哪一个数据块文件。这里判断的逻辑是：

- 选择时间戳大的文件。
- 如果时间戳相同，则选择文件长度大的文件。
- 如果文件长度也相同，则选择在持久化卷上的文件。
- 如果上述三条全都一样，则选择第一个文件。

操作数据块副本方法

在 `FsDatasetImpl` 实现中最重要的一部分就是操作数据块副本的方法，如图 4-28 所示，在 `FsDatasetImpl` 中操作数据块副本的方法会对副本的状态产生影响。这一节我们结合图 4-28 介绍 `FsDatasetImpl` 中涉及数据块副本操作的所有方法。

createTemporary()

`createTemporary()` 方法用于在 `tmp` 文件夹中创建一个新的副本（当创建数据块副本是在数据块复制和集群数据块平衡过程中发起时，副本就会被放置到 `tmp` 目录中并且为 `TEMPORARY` 状态，`TEMPORARY` 状态的副本对于读线程是不可见的）。这个方法是在当前

Datanode 接收其他 Datanode 写数据块的请求时，在 BlockReceiver 的构造方法中调用的。成功创建了 TEMPORARY 副本之后，Datanode 就可以向该副本中写入数据了。

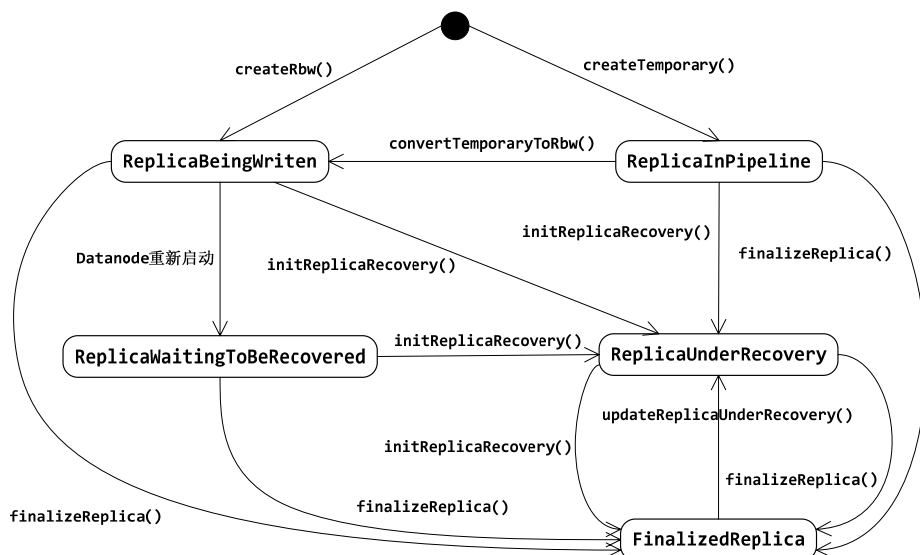


图 4-28 数据块副本状态转移图

createTemporary()方法的代码实现很简单，首先调用 FsVolumeList.getNextVolume()方法获取一个能够保存这个副本的 FsVolumeImpl 对象，然后在这个 FsVolumeImpl 对象上调用 createTmpFile()方法在该数据块副本所在块池的 tmp 目录中创建副本文件，接下来构造 ReplicaInPipeline 对象放入 FsDatasetImpl.volumeMap 中保存，最后返回这个 ReplicaInPipeline 对象。

```

public synchronized ReplicaInPipeline createTemporary(StorageType storageType,
    ExtendedBlock b) throws IOException {
    ReplicaInfo replicaInfo = volumeMap.get(b.getBlockPoolId(), b.getBlockId());
    if (replicaInfo != null) {
        // 如果副本在内存中已经存在，但新创建副本的时间戳更大时
        if (replicaInfo.getGenerationStamp() < b.getGenerationStamp()
            && replicaInfo instanceof ReplicaInPipeline) {
            // 停止对该副本的写操作，并将副本删除
            ((ReplicaInPipeline) replicaInfo)
                .stopWriter(datanode.getDnConf().getXceiverStopTimeout());
            invalidate(b.getBlockPoolId(), new Block[]{replicaInfo});
        } else {
            // 否则直接抛出异常
            throw new ReplicaAlreadyExistsException("Block " + b +
                " already exists in state " + replicaInfo.getState() +
                " and thus cannot be created.");
        }
    }
}

```

```
}  
// 获取一个 FsVolumeImpl 对象来存储该副本  
FsVolumeImpl v = volumes.getNextVolume(storageType, b.getNumBytes());  
// 在该存储目录下的数据块所在块池的 tmp 子目录中创建数据块文件  
File f = v.createTmpFile(b.getBlockPoolId(), b.getLocalBlock());  
// 构造 ReplicaInPipeline 对象记录新创建副本的信息  
ReplicaInPipeline newReplicaInfo = new ReplicaInPipeline(b.getBlockId(),  
    b.getGenerationStamp(), v, f.getParentFile(), 0);  
// 将新构造的 ReplicaInPipeline 对象放入 FsDatasetImpl.volumeMap 中  
volumeMap.add(b.getBlockPoolId(), newReplicaInfo);  
return newReplicaInfo;  
}
```

createRbw()

`createRbw()`方法用于在 `rbw` 文件夹中创建一个副本（当 HDFS 客户端发起写请求而创建一个副本时，这个副本将会被放到 `rbw` 文件夹中，并设置为 RBW 状态，RBW 状态的副本对于读线程是可见的）。这个方法是在当前 `Datanode` 接收来自客户端的写数据请求或者追加写数据请求时，在 `BlockReceiver` 的构造方法中调用的。`createRbw()`方法的实现与 `createTemporary()`是大致相同的，不同的是副本存放在块池目录的 `rbw` 文件夹下，同时增加了 `lazyPersist` 的支持。这里就不再贴出代码了。

recoverRbw()

`recoverRbw()`方法用于恢复一个 RBW 状态的副本，并返回这个副本的信息。这个方法是在 `Datanode` 进行数据流管道恢复操作时，在 `BlockReceiver` 的构造方法中调用的。调用完 `recoverRbw()`方法后，数据流管道中的所有 RBW 状态副本的时间戳和文件长度也就同步了，可以继续通过数据流管道写入新的数据了。

`recoverRbw()`方法主要进行了三个操作：停止副本原先的写入线程，并将当前线程更新为写入线程；对时间戳进行检查，对 `rbw` 数据块的长度进行检查；检查完成之后，更新 `rbw` 副本的时间戳，然后将副本中没有 `ack` 的数据抛弃。

```
public synchronized ReplicaInPipeline recoverRbw(ExtendedBlock b,  
    long newGS, long minBytesRcvd, long maxBytesRcvd)  
    throws IOException {  
    // ...  
    // 停止副本原先的写入线程，并将当前线程更新为写入线程  
    rbw.stopWriter(datanode.getDnConf().getXceiverStopTimeout());  
    rbw.setWriter(Thread.currentThread());  
  
    // 检查时间戳范围，如果不在 b.getGenerationStamp() 至 newGS 之间则抛出异常  
    long replicaGenerationStamp = rbw.getGenerationStamp();  
    if (replicaGenerationStamp < b.getGenerationStamp() ||  
        replicaGenerationStamp > newGS) {  
        throw new ReplicaNotFoundException(  
            ReplicaNotFoundException.UNEXPECTED_GS_REPLICA + b +  
            ". Expected GS range is [" + b.getGenerationStamp() + ", " +
```

```

        newGS + "].");
    }

    // 检查副本的长度, 如果不在 minBytesRcvd 与 maxBytesRcvd 之间则抛出异常
    long bytesAked = rbw.getBytesAked();
    long numBytes = rbw.getNumBytes();
    if (bytesAked < minBytesRcvd || numBytes > maxBytesRcvd) {
        throw new ReplicaNotFoundException("Unmatched length replica " +
            replicaInfo + ": BytesAked = " + bytesAked +
            " BytesRcvd = " + numBytes + " are not in the range of [" +
            minBytesRcvd + ", " + maxBytesRcvd + "].");
    }

    // 将副本中所有没有确认的数据包删除
    if (numBytes > bytesAked) {
        final File replicafile = rbw.getBlockFile();
        truncateBlock(replicafile, rbw.getMetaFile(), numBytes, bytesAked);
        rbw.setNumBytes(bytesAked);
        rbw.setLastChecksumAndDataLen(bytesAked, null);
    }

    // 设置新的时间戳
    bumpReplicaGS(rbw, newGS);

    return rbw;
}

```

convertTemporaryToRbw()

convertTemporaryToRbw()方法用于将 TEMPORARY 状态的副本转换为 RBW 状态。这个方法是在当前 Datanode 从其他 Datanode 成功地接收了复制 (transfer) 的数据块时, 在 BlockReceiver.receiveBlock()方法中调用的。为什么做这个调用呢? 因为在当前 Datanode 从其他 Datanode 接收数据时, 处于 TEMPORARY 状态的副本是不完整的, 并且对于读线程是不可见的。当数据块传输完成后, Datanode 上的数据块完整了, 也就是可以进行读取操作了, 这时就需要将副本状态从 TEMPORARY 转换为 RBW 状态 (只有 RBW 状态的副本对于客户端才是可见的)。

convertTemporaryToRbw()方法首先对副本状态、时间戳以及数据块文件长度进行了检查。如果出现时间戳不等、数据块文件长度不匹配或者副本状态异常等情况, 则直接抛出异常。如果这些判断都正常, 则将 tmp 文件夹中的副本文件移动到 rbw 文件夹下, 然后构造新的 ReplicaBeingWritten 对象并更新 volumeMap。

```

public synchronized ReplicaInPipeline convertTemporaryToRbw(
    final ExtendedBlock b) throws IOException {
    // ...
    final ReplicaInPipeline temp;
    {
        // 检查副本信息是否存在
    }
}

```

Hadoop 2.X HDFS 源码剖析

```
final ReplicaInfo r = volumeMap.get(b.getBlockPoolId(), blockId);
if (r == null) {
    throw new ReplicaNotFoundException(
        ReplicaNotFoundException.NON_EXISTENT_REPLICA + b);
}
// 检查副本状态是否为 TEMPORARY, 否则抛出异常
if (r.getState() != ReplicaState.TEMPORARY) {
    throw new ReplicaAlreadyExistsException(
        "r.getState() != ReplicaState.TEMPORARY, r=" + r);
}
temp = (ReplicaInPipeline)r;
}
// 检查时间戳是否正常
if (temp.getGenerationStamp() != expectedGs) {
    throw new ReplicaAlreadyExistsException(
        "temp.getGenerationStamp() != expectedGs = " + expectedGs
        + ", temp=" + temp);
}

// 检查副本长度, 如果副本长度小于传入的长度, 则抛出异常
final long numBytes = temp.getNumBytes();
if (numBytes < visible) {
    throw new IOException(numBytes + " = numBytes < visible = "
        + visible + ", temp=" + temp);
}
// 检查副本所在存储目录的 FsVolumeImpl 是否存在, 不存在则抛出异常
final FsVolumeImpl v = (FsVolumeImpl)temp.getVolume();
if (v == null) {
    throw new IOException("r.getVolume() = null, temp=" + temp);
}

// 将副本从 tmp 目录移动到 rbw 目录
BlockPoolSlice bpslice = v.getBlockPoolSlice(b.getBlockPoolId());
final File dest = moveBlockFiles(b.getLocalBlock(), temp.getBlockFile(),
    bpslice.getRbwDir());
// 构造新的 ReplicaInfo 对象, 并在 volumeMap 中更新
final ReplicaBeingWritten rbw = new ReplicaBeingWritten(
    blockId, numBytes, expectedGs,
    v, dest.getParentFile(), Thread.currentThread(), 0);
rbw.setBytesAked(visible);
volumeMap.add(b.getBlockPoolId(), rbw);
// 返回这个新构造的 ReplicaBeingWritten 对象
return rbw;
}
```

🔗 append()

append()方法用于对 FINALIZED 状态的副本执行追加写操作。这个方法首先进行副本状

态检查、时间戳检查以及文件长度检查，然后调用私有的 `append()` 方法进行真正的追加写操作。

私有的 `append()` 方法首先调用 `uncacheBlock()` 方法，将当前副本从缓存中移出（`uncacheBlock()` 方法，我们在后面的缓存操作方法小节中介绍）。移出的原因是缓存中保存的数据块是需要确保数据块文件的校验和完全匹配的，如果进行了 `append()` 操作，那么数据块文件的校验和需要重新确认。

调用完 `uncacheBlock()` 方法之后，`append()` 方法会调用 `ReplicaInfo.unlinkBlock()` 方法去除该数据块文件的硬链接。调用这个方法的原因是在 Datanode 升级过程中，会创建数据块文件的快照，但是为了节省磁盘空间，其实是建立了数据块文件的硬链接（例如升级过程中的 `current` 与 `previous` 之间的数据块就是建立了硬链接）。当需要在 `current` 目录中追加写（`append`）一个数据块时，如果修改了 `current` 中的文件，`previous` 中记录的快照文件也修改了，所以这时就需要进行 `unlink` 操作。方法很简单，就是在同一个目录下创建一个临时文件，然后将源文件的内容拷贝到临时文件中，再将临时文件改名成源文件，这样硬链接就解除了。`unlinkBlock()` 的具体实现我们在 `ReplicaInfo` 小节中已经介绍过了，读者可以去 `ReplicaInfo` 小节学习。

`append()` 方法后面的部分就比较简单了，它将数据块文件与校验和文件移动到 `rbw` 目录中，然后构造 `ReplicaBeingWritten` 对象，更新 `volumeMap`，并返回新构造的 `ReplicaBeingWritten` 对象。

```
private synchronized ReplicaBeingWritten append(String bpid,
    FinalizedReplica replicaInfo, long newGS, long estimateBlockLen)
    throws IOException {
    // 如果副本被缓存了，则从缓存中移除该副本
    cacheManager.uncacheBlock(bpid, replicaInfo.getBlockId());
    // 对于有硬链接的副本，先解除硬链接
    replicaInfo.unlinkBlock(1);

    // 构造新的 ReplicaInfo 对象
    File blkfile = replicaInfo.getBlockFile();
    FsVolumeImpl v = (FsVolumeImpl) replicaInfo.getVolume();
    if (v.getAvailable() < estimateBlockLen - replicaInfo.getNumBytes()) {
        throw new DiskOutOfSpaceException("Insufficient space for appending to "
            + replicaInfo);
    }
    File newBlkFile = new File(v.getRbwDir(bpid), replicaInfo.getBlockName());
    File oldmeta = replicaInfo.getMetaFile();
    ReplicaBeingWritten newReplicaInfo = new ReplicaBeingWritten(
        replicaInfo.getBlockId(), replicaInfo.getNumBytes(), newGS,
        v, newBlkFile.getParentFile(), Thread.currentThread(), estimateBlockLen);
    File newmeta = newReplicaInfo.getMetaFile();

    // 将校验和文件移动到 rbw 目录中
    try {
        NativeIO.renameTo(oldmeta, newmeta);
```

```
} catch (IOException e) {
    throw new IOException("Block " + replicaInfo + " reopen failed. " +
        " Unable to move meta file " + oldmeta +
        " to rbw dir " + newmeta, e);
}

// 将数据块文件移动到 rbw 目录中
try {
    NativeIO.renameTo(blkfile, newBlkFile);
} catch (IOException e) {
    try {
        NativeIO.renameTo(newmeta, oldmeta);
    } catch (IOException ex) {
        LOG.warn("Cannot move meta file " + newmeta +
            "back to the finalized directory " + oldmeta, ex);
    }
    throw new IOException("Block " + replicaInfo + " reopen failed. " +
        " Unable to move block file " + blkfile +
        " to rbw dir " + newBlkFile, e);
}

// 更新 volumeMap 中副本的 ReplicaInfo 对象
volumeMap.add(bpid, newReplicaInfo);
v.reserveSpaceForRbw(estimateBlockLen - replicaInfo.getNumBytes());
// 返回构造的 ReplicaInfo 对象
return newReplicaInfo;
}
```

recoverAppend()

`recoverAppend()`方法用于恢复一个失败的追加写操作。`recoverAppend()`方法是在 `Datanode` 进行数据流管道恢复操作时，在 `BlockReceiver` 的构造方法中调用的。调用完 `recoverAppend()` 方法后，数据流管道中所有副本的状态都恢复至 `RBW`，并且副本的时间戳和数据块文件的长度也都一致，可以继续通过数据流管道执行追加写（`append`）操作了。

`recoverAppend()`会首先调用 `recoverCheck()`方法对副本对应的 `RelicaInfo` 对象进行检查。然后判断如果副本状态为 `FINALIZED`，则重新调用 `append()`方法将副本状态恢复为 `RBW`（请参考 `append()`方法分析）；如果副本状态为 `RBW`，则更新副本的时间戳。

```
public synchronized ReplicaInPipeline recoverAppend(ExtendedBlock b,
    long newGS, long expectedBlockLen) throws IOException {
    // 先调用 recoverCheck() 方法对 ReplicaInfo 进行检查
    ReplicaInfo replicaInfo = recoverCheck(b, newGS, expectedBlockLen);

    // 之后更改副本的状态以及时间戳
    if (replicaInfo.getState() == ReplicaState.FINALIZED ) {
        // FINALIZED 状态的副本变为 RBW 状态
        return append(b.getBlockPoolId(), (FinalizedReplica) replicaInfo, newGS,
```

```

        b.getNumBytes());
    } else {
        // RBW 状态的副本，则直接更新时间戳即可
        bumpReplicaGS(replicaInfo, newGS);
        return (ReplicaBeingWritten)replicaInfo;
    }
}

```

这里我们看一下 `recoverCheck()` 方法的实现。这个方法会对副本对应的 `ReplicaInfo` 对象进行检查，主要是检查副本状态、时间戳、数据块文件长度是否正常。

```

private ReplicaInfo recoverCheck(ExtendedBlock b, long newGS,
    long expectedBlockLen) throws IOException {
    ReplicaInfo replicaInfo = getReplicaInfo(b.getBlockPoolId(), b.getBlockId());

    // 检查副本状态，只有 FINALIZED 以及 RBW 状态的副本才能进行追加写操作
    if (replicaInfo.getState() != ReplicaState.FINALIZED &&
        replicaInfo.getState() != ReplicaState.RBW) {
        throw new ReplicaNotFoundException(
            ReplicaNotFoundException.UNFINALIZED_AND_NONRBW_REPLICA + replicaInfo);
    }

    // 检查时间戳，时间戳的范围应当是在原记录的时间戳与新时间戳之间
    long replicaGenerationStamp = replicaInfo.getGenerationStamp();
    if (replicaGenerationStamp < b.getGenerationStamp() ||
        replicaGenerationStamp > newGS) {
        throw new ReplicaNotFoundException(
            ReplicaNotFoundException.UNEXPECTED_GS_REPLICA + replicaGenerationStamp
            + ". Expected GS range is [" + b.getGenerationStamp() + ", " +
            newGS + "].");
    }

    long replicaLen = replicaInfo.getNumBytes();
    if (replicaInfo.getState() == ReplicaState.RBW) {
        ReplicaBeingWritten rbw = (ReplicaBeingWritten)replicaInfo;
        // 如果副本当前处于正在写状态，则更新写线程
        rbw.stopWriter(datanode.getDnConf().getXceiverStopTimeout());
        rbw.setWriter(Thread.currentThread());
        // 确认 bytesRcvd、bytesOnDisk 和 bytesAcks 相同
        if (replicaLen != rbw.getBytesOnDisk())
            || replicaLen != rbw.getBytesAcks()) {
            throw new ReplicaAlreadyExistsException("RBW replica " + replicaInfo +
                "bytesRcvd(" + rbw.getNumBytes() + "), bytesOnDisk(" +
                rbw.getBytesOnDisk() + "), and bytesAcks(" + rbw.getBytesAcks() +
                ") are not the same.");
        }
    }
    // 确认副本长度和期待长度相同，否则抛出异常
    if (replicaLen != expectedBlockLen) {

```

```
        throw new IOException("Corrupted replica " + replicaInfo +  
            " with a length of " + replicaLen +  
            " expected length is " + expectedBlockLen);  
    }  
  
    return replicaInfo;  
}
```

recoverClose()

`recoverClose()` 是在 `Datanode` 进行写入操作的数据流管道恢复时，在 `DataXceiver.writeBlock()` 方法中调用的。`recoverClose()` 方法的实现类似于 `recoverAppend()` 方法，这里不再详细解释了。

finalizeBlock()

`finalizeBlock()` 方法用于提交数据块。当 `Datanode` 完成了一个数据块的写操作并成功接收到数据流管道下游节点的 `ack` 时，会在 `PacketResponder.run()` 方法中调用 `finalizeBlock()` 提交新写入的数据块。

`finalizeBlock()` 方法调用了私有的 `finalizeReplica()` 方法。这个方法首先判断副本的当前状态是不是 `RUR`，如果副本的当前状态是 `RUR`，并且该副本进行恢复操作前的状态是 `FINALIZED`，那么直接将副本的状态转换为 `FINALIZED` 即可；如果是其他状态，则调用 `FsVolumeImpl.addFinalizedBlock()` 方法将数据块文件移动到 `finalize` 目录下。然后构造新的 `FinalizedReplica` 对象并更新 `FsDatasetImpl.volumeMap`。

```
private synchronized FinalizedReplica finalizeReplica(String bpid,  
    ReplicaInfo replicaInfo) throws IOException {  
    FinalizedReplica newReplicaInfo = null;  
    // 如果副本处于恢复状态，并且恢复前的状态为 FINALIZED  
    // 则直接将副本状态转换为 FINALIZED 即可  
    if (replicaInfo.getState() == ReplicaState.RUR &&  
        ((ReplicaUnderRecovery) replicaInfo).getOriginalReplica().getState() ==  
        ReplicaState.FINALIZED) {  
        newReplicaInfo = (FinalizedReplica)  
            ((ReplicaUnderRecovery) replicaInfo).getOriginalReplica();  
    } else {  
        // 对于其他情况，则首先获取 FsVolumeImpl 对象  
        FsVolumeImpl v = (FsVolumeImpl) replicaInfo.getVolume();  
        File f = replicaInfo.getBlockFile();  
        if (v == null) {  
            throw new IOException("No volume for temporary file " + f +  
                " for block " + replicaInfo);  
        }  
  
        // 然后调用 addFinalizedBlock() 方法将数据块移至 finalized 目录下  
        File dest = v.addFinalizedBlock(  
            bpid, replicaInfo, f, replicaInfo.getBytesReserved());  
    }
```



```

newReplicaInfo = new FinalizedReplica(replicaInfo, v, dest.getParentFile());

if (v.isTransientStorage()) {
    ramDiskReplicaTracker.addReplica(bpid, replicaInfo.getBlockId(), v);
    datanode.getMetrics().addRamDiskBytesWrite(replicaInfo.getNumBytes());
}
}
// 更新 volumeMap 中的副本信息
volumeMap.add(bpid, newReplicaInfo);
// 返回新构造的 ReplicaInfo 对象
return newReplicaInfo;
}

```

✎ unfinalizeBlock()

`unfinalizeBlock()`方法用于删除 TEMPORARY 状态的副本,也就是复制失败的副本。这个方法直接检查副本状态,删除数据块文件以及对应的校验和文件,然后从 `FsDatasetImpl.volumeMap` 中移除副本的 `ReplicaInfo`。比较简单,这里就不多解释了。

✎ invalidate()

`invalidate()`方法也是用来删除副本的,与 `unfinalizeBlock()`方法不同的是, `invalidate()`方法是用于删除 RBW 状态以及 FINALIZED 状态的副本的。因为副本有可能被短路读取 (RBW 状态以及 FINALIZED 状态的副本对客户端是可见的),并且可能存储在 Datanode 与客户端的短路读取共享内存中 (请参考第5章),所以 `invalidate()`方法需要从短路读取的共享内存中移出副本的引用。然后 `invalidate()`方法会将副本从 Datanode 缓存中删除,最后 `invalidate()`方法会调用异步的磁盘文件删除方法 `FsDatasetAsyncDiskService.deleteAsync()`将副本的数据块文件从磁盘上删除。

异步的磁盘操作降低了方法的响应时间,底层就是使用线程池进行异步操作的。这是一种非常典型的提高系统性能的方式,请读者积累。请参考 `FsDatasetAsyncDiskService` 的代码实现。

我们看一下 `invalidate()`方法的代码实现。

```

public void invalidate(String bpid, Block invalidBlks[]) throws IOException {
    final List<String> errors = new ArrayList<String>();
    for (int i = 0; i < invalidBlks.length; i++) {
        final File f;
        final FsVolumeImpl v;
        synchronized (this) {
            final ReplicaInfo info = volumeMap.get(bpid, invalidBlks[i]);
            // ...判断 ReplicaInfo 信息是否合法,这里省略
            // 从 volumeMap 中移除副本信息
            volumeMap.remove(bpid, invalidBlks[i]);
        }
        // ...
        // 如果副本正在被短路读取,并且副本信息存储在短路读取的共享内存中,则删除
    }
}

```

```
datanode.getShortCircuitRegistry().processBlockInvalidation(  
    new ExtendedBlockId(invalidBlks[i].getBlockId(), bpid));  
  
// 如果副本在 Datanode 中被缓存, 则从缓存中删除当前副本  
cacheManager.uncacheBlock(bpid, invalidBlks[i].getBlockId());  
  
// 调用异步的磁盘文件删除功能, 删除数据块文件以及校验文件  
asyncDiskService.deleteAsync(v, f,  
    FsDatasetUtil.getMetaFile(f, invalidBlks[i].getGenerationStamp()),  
    new ExtendedBlock(bpid, invalidBlks[i]),  
    dataStorage.getTrashDirectoryForBlockFile(bpid, f));  
}  
if (!errors.isEmpty()) {  
    StringBuilder b = new StringBuilder("Failed to delete ")  
        .append(errors.size()).append(" (out of ").append(invalidBlks.length)  
        .append(") replica(s):");  
    for(int i = 0; i < errors.size(); i++) {  
        b.append("\n").append(i).append(" ").append(errors.get(i));  
    }  
    throw new IOException(b.toString());  
}  
}
```

initReplicaRecovery()

当客户端写文件时异常退出, 且未能正常关闭文件时, 文件的最后一个数据块的所有副本可能处于不同的状态。当客户端再次打开这个文件进行操作时, 就需要先对最后一个数据块进行块恢复操作 (BlockRecovery), 使得该数据块的所有副本状态一致, 然后才能进行后续的操作。

Namenode 初始化块恢复操作时, 会先选定一个主恢复数据节点 (Primary Datanode) 来控制整个数据块恢复流程。然后 Namenode 会通过心跳携带 LeaseRecovery 指令给主恢复数据节点开始块恢复操作, 主恢复数据节点会通过调用 InterDatanodeProtocol 接口与数据流管道中的其他 Datanode 通信, 控制整个数据块恢复流程。

主恢复节点首先调用 InterDatanodeProtocol.initReplicaRecovery()方法获取数据流管道中各个 Datanode 的副本恢复信息, 然后从所有副本中选取一个最好的状态作为所有副本恢复操作的目标状态。最后主恢复节点会调用 InterDatanodeProtocol.updateReplicaUnderRecovery()方法同步所有 Datanode 上该副本的状态至目标状态。

FsDatasetImpl.initReplicaRecovery()方法就用于获取当前 Datanode 上指定数据块副本的状态。这个方法会返回一个 ReplicaRecoveryInfo 对象, 它包含了该副本在恢复前的信息和状态。这个方法的实现比较简单, 首先从 volumeMap 中获取该副本的信息, 如果有写线程正在写这个副本, 则停止写线程以保证数据块的恢复过程不受写线程的影响。之后检查副本的时间戳以及恢复标识 (recoveryId), 这里的恢复标识就是 Namenode 为该数据块分配的新的时间戳。当恢复操作完成后, 副本的时间戳将会被设置为恢复标识。如果当前副本的时间戳大于恢复

标识，则当前恢复操作是一个过期的恢复操作，抛出异常。检查完时间戳和恢复标识之后，`initReplicaRecovery()`方法将会构造一个 `ReplicaUnderRecovery` 对象，表明当前副本正在进行副本恢复操作，副本状态为 `RUR`，然后使用这个 `ReplicaUnderRecovery` 更新 `volumeMap` 中副本的信息。

```
static ReplicaRecoveryInfo initReplicaRecovery(String bpid, ReplicaMap map,
    Block block, long recoveryId, long xceiverStopTimeout) throws IOException {
    // 从 volumeMap 中获取副本信息
    final ReplicaInfo replica = map.get(bpid, block.getBlockId());

    // 确认副本存在
    if (replica == null) {
        return null;
    }

    // 如果有写线程则停止写线程
    if (replica instanceof ReplicaInPipeline) {
        final ReplicaInPipeline rip = (ReplicaInPipeline)replica;
        rip.stopWriter(xceiverStopTimeout);
        // 确认接收的数据大于可见的数据
        if (rip.getBytesOnDisk() < rip.getVisibleLength()) {
            throw new IOException("THIS IS NOT SUPPOSED TO HAPPEN:"
                + " getBytesOnDisk() < getVisibleLength(), rip=" + rip);
        }
        // 检查副本文件
        checkReplicaFiles(rip);
    }

    // 检查副本时间戳
    if (replica.getGenerationStamp() < block.getGenerationStamp()) {
        throw new IOException(
            "replica.getGenerationStamp() < block.getGenerationStamp(), block="
            + block + ", replica=" + replica);
    }

    // 检查 recoveryId，确认不是一个过期的恢复操作
    if (replica.getGenerationStamp() >= recoveryId) {
        throw new IOException("THIS IS NOT SUPPOSED TO HAPPEN:"
            + " replica.getGenerationStamp() >= recoveryId = " + recoveryId
            + ", block=" + block + ", replica=" + replica);
    }

    // 构造 ReplicaUnderRecovery 对象，如果已经存在 RUR 对象，则更新 recoveryId
    final ReplicaUnderRecovery rur;
    if (replica.getState() == ReplicaState.RUR) {
        rur = (ReplicaUnderRecovery)replica;
        // 注意，这里要判断这个恢复操作是否是一个过期或者重复的操作
        if (rur.getRecoveryID() >= recoveryId) {
```

```
        throw new RecoveryInProgressException(
            "rur.getRecoveryID() >= recoveryId = " + recoveryId
            + ", block=" + block + ", rur=" + rur);
    }
    final long oldRecoveryID = rur.getRecoveryID();
    rur.setRecoveryID(recoveryId);
}
// ReplicaUnderRecovery 对象不存在, 则构造 RUR 对象, 并更新 volumeMap
else {
    rur = new ReplicaUnderRecovery(replica, recoveryId);
    map.add(bpid, rur);
}
// 返回副本恢复前的信息
return rur.createInfo();
}
```

updateReplicaUnderRecovery()

主恢复节点调用 `InterDatanodeProtocol.initReplicaRecovery()` 方法后, 将会获得数据流管道中所有恢复数据块的副本状态, 主恢复节点会在所有副本中选取一个最好的状态作为恢复操作的目标状态。判断副本状态的标准是 `FINALIZED > RBW > RWR > RUR > TEMPORARY`, 也就是副本的持久化程度越高则副本状态越好。如果所有副本中最好的状态是 `FINALIZED`, 那么就以这个 `FINALIZED` 副本长度为目标, 其他和这个副本不一致的都排除在恢复操作外; 如果最好的状态是 `RBW` 或者 `RWR`, 那么就选择所有副本中数据块文件长度最短的副本作为目标副本。

得到了副本恢复操作的目标状态以及参与恢复的 `Datanode` 列表后, 主恢复节点会对列表中的所有 `Datanode` 调用 `InterDatanodeProtocol.updateReplicaUnderRecovery()` 方法同步该数据节点上副本的状态至目标状态。

`FsDatasetImpl.updateReplicaUnderRecovery()` 方法就用于将当前数据节点上副本的状态同步至目标状态。`updateReplicaUnderRecovery()` 方法会首先检查副本是不是 `RUR` 状态, 检查副本在磁盘上的长度是否正常, 检查副本的数据块文件和校验文件是否匹配等。然后 `updateReplicaUnderRecovery()` 方法会调用私有的 `updateReplicaUnderRecovery()` 将当前副本的时间戳更新为恢复标识 (`recoveryId`), 将副本的数据块文件长度截短为目标长度并重新计算校验和。最后 `updateReplicaUnderRecovery()` 方法会将副本由 `RUR` 状态更改为 `FINALIZED` 状态, 并更新 `FsDatasetImpl.volumeMap`。至此, 恢复操作完成, 数据流管道中的所有副本状态一致。

```
public synchronized String updateReplicaUnderRecovery(
    final ExtendedBlock oldBlock,
    final long recoveryId,
    final long newlength) throws IOException {
    // 获取副本信息
    final String bpid = oldBlock.getBlockPoolId();
    final ReplicaInfo replica = volumeMap.get(bpid, oldBlock.getBlockId());
```

```

// 检查副本信息是否存在
if (replica == null) {
    throw new ReplicaNotFoundException(oldBlock);
}

// 判断副本状态, 如果不是 RUR 则抛出异常
if (replica.getState() != ReplicaState.RUR) {
    throw new IOException("replica.getState() != " + ReplicaState.RUR
        + ", replica=" + replica);
}

// 检查副本长度是否正常
if (replica.getBytesOnDisk() != oldBlock.getNumBytes()) {
    throw new IOException("THIS IS NOT SUPPOSED TO HAPPEN:"
        + " replica.getBytesOnDisk() != block.getNumBytes(), block="
        + oldBlock + ", replica=" + replica);
}

// 检查副本的数据块文件和校验文件是否匹配
checkReplicaFiles(replica);

// 将副本状态更新为目标状态
final FinalizedReplica finalized = updateReplicaUnderRecovery(oldBlock
    .getBlockPoolId(), (ReplicaUnderRecovery) replica, recoveryId, newlength);
assert finalized.getBlockId() == oldBlock.getBlockId()
    && finalized.getGenerationStamp() == recoveryId
    && finalized.getNumBytes() == newlength
    : "Replica information mismatched: oldBlock=" + oldBlock
        + ", recoveryId=" + recoveryId + ", newlength=" + newlength
        + ", finalized=" + finalized;

// 检查副本的数据块文件和校验文件是否匹配
checkReplicaFiles(finalized);

// 返回存储 ID
return getVolume(new ExtendedBlock(bpid, finalized)).getStorageID();
}

// 私有的 updateReplicaUnderRecovery() 方法
private FinalizedReplica updateReplicaUnderRecovery(
    String bpid,
    ReplicaUnderRecovery rur,
    long recoveryId,
    long newlength) throws IOException {
    // 检查恢复标识, 如果是过期的恢复操作, 则抛出异常
    if (rur.getRecoveryID() != recoveryId) {

```

```
        throw new IOException("rur.getRecoveryID() != recoveryId = " + recoveryId
            + ", rur=" + rur);
    }

    // 更新副本时间戳到 recoveryId
    bumpReplicaGS(rur, recoveryId);

    // 更新副本数据块文件的长度
    final File replicafile = rur.getBlockFile();
    if (rur.getNumBytes() < newlength) {
        throw new IOException("rur.getNumBytes() < newlength = " + newlength
            + ", rur=" + rur);
    }
    if (rur.getNumBytes() > newlength) {
        // 如果数据块文件的长度大于目标长度
        // 则调用 truncate() 方法截短数据块文件, 并重新计算校验和
        rur.unlinkBlock(1);
        truncateBlock(replicafile, rur.getMetaFile(), rur.getNumBytes(), newlength);
        rur.setNumBytes(newlength);
    }

    // 将副本状态由 RUR 改为 FINALIZED, 并更新 volumeMap
    return finalizeReplica(bpid, rur);
}
```

缓存操作方法

Hadoop 2.3.0 中加入了集中式缓存管理功能 (HDFS Centralized Cache Management), 用户可以通过 “hdfs cacheadmin” 命令或者 HDFS API 显式地将 HDFS 上某个文件或者目录放到集中式缓存中。集中式缓存由分布在 Datanode 上的堆外内存组成, 同时被 Namenode 统一管理。

FsDatasetImpl 中与缓存相关的方法大多调用了 FsDatasetCache 对应的方法, 本节我们首先学习 FsDatasetCache 类的实现, 然后介绍 FsDatasetImpl.cache() 以及 FsDatasetImpl.uncache() 方法的实现。

📌 FsDatasetCache 类

FsDatasetCache 类提供了管理 Datanode 上缓存数据块的功能, FsDatasetCache 类会通过调用 mmap(2) 和 mlock(2) 这两个系统调用在 Datanode 的内存中缓存数据块。FsDatasetCache 类还定义了 mappableBlockMap 字段保存当前 Datanode 上所有缓存数据块的信息, 如下代码所示, mappableBlockMap 字段是一个 blockId 与 Value 对象的映射。这里的 Value 类是 FsDatasetCache 的内部类, 有 state 和 mappableBlock 两个字段。

```
public class FsDatasetCache {

    private final HashMap<ExtendedBlockId, Value> mappableBlockMap =
        new HashMap<ExtendedBlockId, Value>();
```

```
private static final class Value {
    final State state;
    final MappableBlock mappableBlock;
    // ...
}
}
```

- **state** 字段：保存当前数据块的缓存状态，是 **State** 类型的。**State** 是 **FsDatasetCache** 的内部枚举类，用于描述缓存数据块可能包括的 4 种状态。
 - **CACHING**：正在进行缓存的状态。由于数据块的缓存操作是异步执行的（请参考 **cache()** 方法分析），所以会存在缓存中状态。
 - **CACHING_CANCELLED**：被取消缓存操作的数据块。
 - **CACHED**：已经完成缓存操作的数据块。
 - **UNCACHING**：从缓存中移除的数据块。需要注意的是，这里似乎没有 **UNCACHED** 状态，因为如果不在 **mappableBlockMap** 中，自然就是 **UNCACHED** 状态了。
- **mappableBlock**：保存在 **Datanode** 内存中缓存的 **HDFS** 数据块，是 **MappableBlock** 类型的。

我们再来看一下 **MappableBlock** 这个类。如下代码所示，**MappableBlock** 定义了一个非常重要的字段 **mmap** 保存缓存中的数据块文件。**mmap** 字段是 **MappedByteBuffer** 类型的，**MappedByteBuffer** 是 **JDK** 提供的直接字节缓冲区，其内容是文件的内存映射区域，在这里就是数据块文件在内存中的映射。

```
public class MappableBlock implements Closeable {
    private MappedByteBuffer mmap;
    // ...
}
```

MappableBlock.mmap 字段是在 **MappableBlock.load()** 方法中初始化的，**MappableBlock.load()** 方法会调用 **Java NIO** 框架的 **FileChannel.map()** 操作将数据块文件映射到内存中，然后获取映射对象 **MappedBuffer**。之后我们就可以通过 **MappedBuffer** 对象对内存进行读写操作了，操作的结果会由操作系统负责 **flush** 到数据块文件中。由于数据块文件的内容都在内存中，所以读取的速度会大大加快。**load()** 方法之后会调用 **mlock** 操作将内存中的这段区域锁住，以防止操作系统将这段内存交换到 **swap** 中。整个映射操作完成之后，**load()** 方法会调用 **verifyChecksum()** 方法对映射到内存中的数据块进行校验操作，这样 **DFSCClient** 就可以通过零拷贝（**zero-copy**）方法直接读取内存中的数据块而不需要校验了（请参考第 5 章）。校验成功之后，**load()** 方法会构造 **MappableBlock** 对象并返回。**load()** 方法的代码如下：

```
public static MappableBlock load(long length,
    FileInputStream blockIn, FileInputStream metaIn,
    String blockFileName) throws IOException {
    MappableBlock mappableBlock = null;
    MappedByteBuffer mmap = null;
    FileChannel blockChannel = null;
    try {
```

```
blockChannel = blockIn.getChannel();
if (blockChannel == null) {
    throw new IOException("Block InputStream has no FileChannel.");
}
// 调用 FileChannel.map() 方法将数据块文件映射到内存中
mmap = blockChannel.map(MapMode.READ_ONLY, 0, length);
// 调用 mlock 系统调用将内存锁定
NativeIO.POSIX.getCacheManipulator().mlock(blockFileName, mmap, length);
// 验证内存中缓存的数据块的校验和正确
verifyChecksum(length, metaIn, blockChannel, blockFileName);
// 构造 MappableBlock 对象
mappableBlock = new MappableBlock(mmap, length);
} finally {
    IOUtils.closeQuietly(blockChannel);
    if (mappableBlock == null) {
        if (mmap != null) {
            NativeIO.POSIX.munmap(mmap); // unmapping also unlocks
        }
    }
}
// 返回新构造的 MappableBlock 对象
return mappableBlock;
}
```

FsDatasetCache 类的 cacheBlock() 和 uncacheBlock()方法我们在后面两个小节中介绍。

▣ FsDatasetImpl.cache()方法

数据块缓存操作的入口是 FsDatasetImpl.cache()方法，这个方法最终调用了私有的 cacheBlock()方法。cacheBlock()方法会首先获取副本的 blockFileName、length、genstamp 以及对应的 FsVolumeImpl 的 cacheExecutor 对象，然后调用 FsDatasetCache.cacheBlock()方法完成数据块缓存操作。

如下代码所示，FsDatasetCache.cacheBlock()方法会在 FsDatasetCache.mappableBlockMap 字段中加入新的数据块映射关系，并将缓存数据块状态设置为 CACHING，之后调用数据块所在 FsVolumeImpl 的线程池异步地执行加载任务——CachingTask。CachingTask 的逻辑比较简单，就是通过调用 MappableBlock.load()方法将数据块文件映射到内存中，然后更新缓存数据块状态为 CACHED，并且在 FsDatasetCache.mappableBlockMap 字段中更新 mappableBlock 的引用。当执行完上述操作之后，CachingTask 还会调用 processBlockMlockEvent()方法，将短路读取共享内存中当前数据块副本的槽位设置为可锚定 (anchorable)，这样本地的 DFSClient 就可以通过零拷贝 (zero-copy) 的方式读取数据块而无须校验了 (请参考第 5 章)。CachingTask.run()方法的代码如下：

```
try {
    // 调用 load() 方法将数据块缓存到内存中
    mappableBlock = MappableBlock.
        load(length, blockIn, metaIn, blockFileName);
```



```

    } catch (ChecksumException e) {
        return; // 如果出现校验和错误，则直接返回
    } catch (IOException e) {
        return;
    }
}
synchronized (FsDatasetCache.this) {
    Value value = mappableBlockMap.get(key);
    if (value.state == State.CACHING_CANCELLED) {
        mappableBlockMap.remove(key);
        LOG.warn("Caching of " + key + " was cancelled.");
        return;
    }
    // 将数据块的缓存状态设置为 CACHED，并放入 mappableBlockMap 字段中保存
    mappableBlockMap.put(key, new Value(mappableBlock, State.CACHED));
}
// 将短路读取共享内存中当前数据块的槽位设置为可锚定
dataset.datanode.getShortCircuitRegistry().processBlockMlockEvent(key);
numBlocksCached.addAndGet(1);
dataset.datanode.getMetrics().incrBlocksCached(1);
success = true;

```

这里需要注意的是，CachingTask 是由线程池异步执行的，这个线程池不由 FsDatasetCache 管理，而是每个 FsVolumeImpl 都有一个单独的线程池 cacheExecutor，用于执行当前 FsVolumeImpl 上数据块的缓存任务（CachingTask）。

🔪 FsDatasetImpl.uncache()方法

FsDatasetImpl.uncache() 方法用于将数据块从缓存中移出，uncache() 方法调用了 FsDatasetCache.uncacheBlock() 方法完成缓存操作。

FsDatasetCache.uncacheBlock() 的代码如下，需要考虑数据块的缓存状态。

```

switch (prevValue.state) {
case CACHING:
    mappableBlockMap.put(key,
        new Value(prevValue.mappableBlock, State.CACHING_CANCELLED));
    break;
case CACHED:
    mappableBlockMap.put(key,
        new Value(prevValue.mappableBlock, State.UNCACHING));
    if (deferred) {
        deferredUncachingExecutor.schedule(
            new UncachingTask(key, revocationMs,
                revocationPollingMs, TimeUnit.MILLISECONDS);
        } else {
            uncachingExecutor.execute(new UncachingTask(key, 0));
        }
    }
    break;
default:

```

```
numBlocksFailedToUncache.incrementAndGet();
break;
}
```

uncached()方法的逻辑如下:

- 如果数据块在 mappableBlockMap 映射中没有记录,那么就不需要进行 uncached 操作了。
- 如果是 CACHING 状态,也就是 CachingTask 正在异步执行过程中,则将缓存状态直接设置为 CACHING_CANCELLED。当 CachingTask 发现当前 MappableBlock 的状态更改为 CACHING_CANCELLED 时,就会从 mappableBlockMap 中删除数据块信息。这里要特别注意方法之间的同步关系。由于 uncached()和 CachingTask.run()方法中对于 mappableBlockMap 字段的修改部分都添加了锁操作,所以这里不会存在语义不同的问题,请参考 CachingTask.run()代码。

```
synchronized (FsDatasetCache.this) {
    Value value = mappableBlockMap.get(key);
    Preconditions.checkNotNull(value);
    Preconditions.checkState(value.state == State.CACHING ||
                             value.state == State.CACHING_CANCELLED);
    if (value.state == State.CACHING_CANCELLED) {
        // 数据块被取消缓存,则从 mappableBlockMap 中移除即可
        mappableBlockMap.remove(key);
        return;
    }
    mappableBlockMap.put(key, new Value(mappableBlock, State.CACHED));
}
```

- 如果是 CACHED 状态,那么构造一个移出缓存任务——UncachingTask。UncachingTask 会释放数据块在内存中的映射区域,然后从 MappableBlockMap 中删除对应的记录。这里我们可以观察到,所有对于 mappableBlockMap 的操作都是加锁的。请参考 UncachingTask.run()代码。

```
public void run() {
    Value value;
    // 获取数据块对应的缓存信息
    synchronized (FsDatasetCache.this) {
        value = mappableBlockMap.get(key);
    }

    // 释放数据块文件在内存中的映射区域,也就是释放 mappableBlock
    IOUtils.closeQuietly(value.mappableBlock);

    // 从 mappableBlockMap 映射中删除当前数据块的记录
    synchronized (FsDatasetCache.this) {
        mappableBlockMap.remove(key);
    }
    long newUsedBytes =
        usedBytesCount.release(value.mappableBlock.getLength());
    numBlocksCached.addAndGet(-1);
}
```

```

        dataset.datanode.getMetrics().incrBlocksUncached(1);
    }
}

```

数据块汇报相关方法 (done)

FsDatasetImpl 的 `getBlockReports()` 和 `getCacheReport()` 方法用于支持 Datanode 的块汇报以及缓存汇报操作。由于 Hadoop 2.X 中引入了 Federation 机制, 所以 Datanode 需要向 HDFS 集群中的所有 Namenode 发送块汇报。

getBlockReports()

我们知道 Datanode 在启动时会通过 `DatanodeProtocol.blockReport()` 方法向 Namenode 汇报数据节点上存储的该命名空间的所有数据块副本的信息, 也就是 Namenode 对应的块池在当前 Datanode 上保存的所有数据块副本的信息。

Datanode 会调用 `FsDatasetImpl.getBlockReports()` 方法获取指定块池在当前 Datanode 上保存的所有副本的信息, 这里的块池信息是由 `getBlockReports()` 方法的参数 `bpid` 指定的。需要注意的是, 块汇报只包含 FINALIZED 状态以及 UnderConstruction (包括 RBW、RWR、RUR) 状态的副本, 不包含 TEMPORARY 状态的副本。同时由于 Hadoop 2.6 版本中引入了异构的存储目录, 块汇报的单位也由整个 Datanode 改为 Datanode 的存储目录, 所以 `getBlockReports()` 方法的返回值是一个 Map 对象, 这个 Map 保存了每个存储目录的 `DatanodeStorage` 对象 (通过 `FsVolumeImpl.toDataStorage()` 方法获得) 与这个存储目录保存的所有副本信息之间的映射。`getBlockReports()` 方法的实现非常简单, 就是遍历 `FsDatasetImpl.volumeMap` 字段中保存的所有副本, 将 FINALIZED 状态以及 RBW、RWR、RUR 状态的副本放入存储目录对应的集合中, 然后返回。

```

public Map<DatanodeStorage, BlockListAsLongs> getBlockReports(String bpid) {

    // 构造返回值对象
    Map<DatanodeStorage, BlockListAsLongs> blockReportsMap =
        new HashMap<DatanodeStorage, BlockListAsLongs>();
    // 保存所有 FINALIZED 副本信息的 Map 对象
    Map<String, ArrayList<ReplicaInfo>> finalized =
        new HashMap<String, ArrayList<ReplicaInfo>>();
    // 保存所有 UnderConstruction 副本信息的 Map 对象
    Map<String, ArrayList<ReplicaInfo>> uc =
        new HashMap<String, ArrayList<ReplicaInfo>>();

    // 初始化 finalized 集合以及 uc 集合
    for (FsVolumeSpi v : volumes.volumes) {
        finalized.put(v.getStorageID(), new ArrayList<ReplicaInfo>());
        uc.put(v.getStorageID(), new ArrayList<ReplicaInfo>());
    }

    synchronized(this) {
        // 遍历 FsDatasetImpl.volumeMap 字段中保存的所有副本信息
        for (ReplicaInfo b : volumeMap.replicas(bpid)) {

```

```
switch(b.getState()) {
    case FINALIZED: // 如果是 FINALIZED 状态, 则加入 finalized 集合中
        // 将副本信息放入对应存储目录的集合中
        finalized.get(b.getVolume().getStorageID()).add(b);
        break;
    case RBW:
    case RWR:
        uc.get(b.getVolume().getStorageID()).add(b);
        break;
    case RUR:// RBW、RWR、RUR 状态, 加入 uc 集合中
        ReplicaUnderRecovery rur = (ReplicaUnderRecovery)b;
        // 将副本信息放入对应存储目录的集合中
        uc.get(rur.getVolume().getStorageID()).add(rur.getOriginalReplica());
        break;
    case TEMPORARY:
        break;
    default:
        assert false : "Illegal ReplicaInfo state.";
}
}
}

// 通过 finalized 以及 uc 集合, 构造返回值对象 blockReportsMap
for (FsVolumeSpi v : volumes.volumes) {
    ArrayList<ReplicaInfo> finalizedList = finalized.get(v.getStorageID());
    ArrayList<ReplicaInfo> ucList = uc.get(v.getStorageID());
    blockReportsMap.put(((FsVolumeImpl) v).toDatanodeStorage(),
        new BlockListAsLongs(finalizedList, ucList));
}

// 返回 blockReportsMap 对象
return blockReportsMap;
}
```

getCacheReport()

getCacheReport() 方法与 getBlockReports() 方法一样, 用于在 Datanode 通过 DatanodeProtocol.cacheReport() 向 Namenode 进行缓存汇报时, 获取对应块池在当前 Datanode 上缓存的所有数据块副本的信息。

getCacheReport() 方法调用了 FsDatasetCache.getCachedBlocks() 方法, getCachedBlocks() 会遍历 FsDatasetCache.mappableBlockMap 字段中保存的所有缓存数据块, 将状态为 CACHED 的缓存数据块保存到一个集合中, 然后返回。

```
synchronized List<Long> getCachedBlocks(String bpid) {
    List<Long> blocks = new ArrayList<Long>();

    // 遍历 FsDatasetCache.mappableBlockMap 字段中保存的所有缓存数据块
    for (Iterator<Entry<ExtendedBlockId, Value>> iter =
        mappableBlockMap.entrySet().iterator(); iter.hasNext(); ) {
```

```

Entry<ExtendedBlockId, Value> entry = iter.next();
if (entry.getKey().getBlockPoolId().equals(bpid)) {
    // 如果缓存数据块的状态为 CACHED, 则加入 blocks 集合中
    if (entry.getValue().state.shouldAdvertise()) {
        blocks.add(entry.getKey().getBlockId());
    }
}
}

// 返回 blocks 集合
return blocks;
}

```

4.4 BlockPoolManager

我们知道在 HDFS Federation 部署中, 一个 HDFS 集群可以配置多个命名空间 (Namespace), 每个 Datanode 都会存储多个块池的数据块。所以在 Datanode 实现中, 定义了 BlockPoolManager 类来管理 Datanode 上的所有块池, Datanode 的其他模块对块池的操作都必须通过 BlockPoolManager 执行, 每个 Datanode 都有一个 BlockPoolManager 的实例。

BlockPoolManager 逻辑结构图如图 4-29 所示。由于在 HDFS Federation 部署中, 一个 Datanode 会保存多个块池的数据块, 所以 BlockPoolManager 会拥有多个 BPOfferService 对象, 每个 BPOfferService 对象都封装了对单个块池操作的 API。同时, 由于在 HDFS HA 部署中, 每个命名空间又会同时拥有两个 Namenode, 一个作为活动的 (Active) Namenode, 另一个作为热备的 (Standby) Namenode, 所以每个 BPOfferService 都会包含两个 BPServiceActor 对象, 每个 BPServiceActor 对象都封装了与该命名空间中单个 Namenode 的操作, 包括定时向这个 Namenode 发送心跳 (heartbeat)、增量块汇报 (blockReceivedAndDeleted)、全量块汇报 (blockreport)、缓存块汇报 (cacheReport), 以及执行 Namenode 通过心跳/块汇报响应传回的名字节点指令等操作。

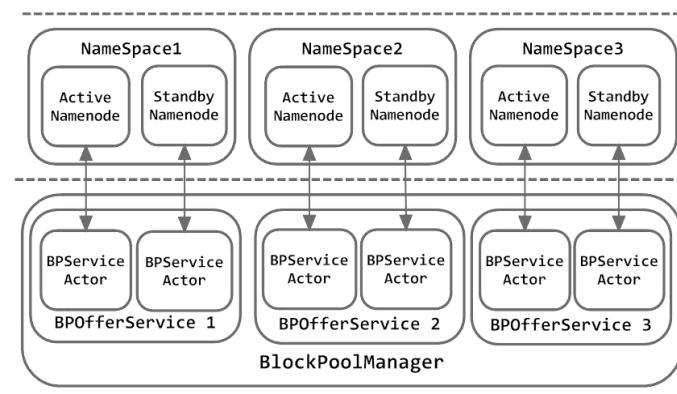


图 4-29 BlockPoolManager 逻辑结构图

本节主要介绍 BlockPoolManager 的具体实现，我们按照 BPSERVICEActor、BPOfferService、BlockPoolManager 这个顺序依次介绍。

4.4.1 BPSERVICEActor 实现

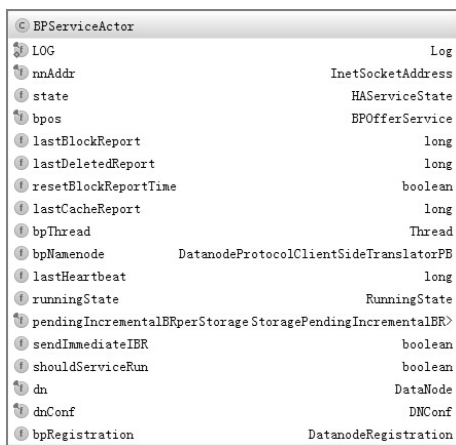
在 BlockPoolManager 的实现中，使用 BPSERVICEActor 类负责与一个 Namenode 通信。每个 BPSERVICEActor 的实例都是一个独立运行的线程，这个线程主要实现了以下功能。

- 与 Namenode 进行第一次握手，获取命名空间的信息。
- 向 Namenode 注册当前 Datanode。
- 定期向 Namenode 发送心跳、增量块汇报、全量块汇报以及缓存块汇报。
- 执行 Namenode 传回的名字节点指令。

这一节我们就学习 BPSERVICEActor 的实现。BPSERVICEActor 中的很多操作都涉及与 BPOfferService 对象的交互，请读者结合 BPOfferService 实现小节一起学习。

1. BPSERVICEActor 的字段

我们首先看一下 BPSERVICEActor 定义了哪些字段。如图 4-30 所示，BPSERVICEActor 定义了以下字段。



LOG	Log
nnAddr	InetSocketAddress
state	HASERVICEState
bpos	BPOfferService
lastBlockReport	long
lastDeletedReport	long
resetBlockReportTime	boolean
lastCacheReport	long
bpThread	Thread
bpNamenode	DatanodeProtocolClientSideTranslatorPB
lastHeartbeat	long
runningState	RunningState
pendingIncrementalBReportStorageStoragePendingIncrementalBReport	
sendImmediateIBR	boolean
shouldServiceRun	boolean
dn	Datanode
dnConf	DNConf
bpRegistration	DatanodeRegistration

图 4-30 BPSERVICEActor 定义的字段

- nnAddr: 当前 BPSERVICEActor 对应的 Namenode 的地址。
- state: 当前 BPSERVICEActor 对应的 Namenode 的状态。
- bpos: 管理当前 BPSERVICEActor 对象的 BPOfferService 对象的引用。
- lastBlockReport、lastDeletedReport、lastCacheReport、lastHeartbeat: 记录上一次数据块汇报、增量块汇报、缓存块汇报以及心跳的时间。
- bpThread: 当前的工作线程，是 BPSERVICEActor 主逻辑的执行线程。
- bpNamenode: 向 Namenode 发送 RPC 请求的代理，Datanode 向 Namenode 发起

DatanodeProtocolRPC 调用都是通过这个引用进行的。

- dn 和 dnConf: Datanode 对象的引用, 以及 Datanode 的配置。
- bpRegistration: 用于记录 Datanode 的注册信息。
- shouldServiceRun: 标志位, 用来指明当前 BPSERVICEActor 的状态, true 为运行状态, false 为停止状态, 这个标志位是用于控制 bpThread 工作线程的。
- runningState: 当前 BPSERVICEActor 的状态, BPSERVICEActor 的状态有 CONNECTING、INIT_FAILED、RUNNING、EXITED、FAILED, 初始状态是 CONNECTING。
- pendingIncrementalBlockStorage: 用于保存两次块汇报之间 Datanode 存储数据块的变化。

2. 构造方法与工作线程控制方法

了解了 BPSERVICEActor 定义的字段, 我们再来看 BPSERVICEActor 的构造方法。BPSERVICEActor 的构造方法比较简单, 通过传入基本参数初始化对应字段即可。

```
BPSERVICEActor(InetSocketAddress nnAddr, BPOfferService bpos) {
    this.bpos = bpos;
    this.dn = bpos.getDataNode();
    this.nnAddr = nnAddr;
    this.dnConf = dn.getDnConf();
}
```

start()、stop()、join()方法用于启动、停止与等待 BPSERVICEActor 的 bpThread 工作线程线程。代码如下:

```
// start() 方法只能由 BPOfferService 调用
void start() {
    if ((bpThread != null) && (bpThread.isAlive())) {
        // bpThread 线程已经启动, 则直接返回
        return;
    }
    // 构造 bpThread 并以守护线程的方式启动
    bpThread = new Thread(this, formatThreadName());
    bpThread.setDaemon(true); // needed for JUnit testing
    bpThread.start();
}

//stop() 方法只能由 blockPoolManager 调用
void stop() {
    // 将 shouldServiceRun 赋值为 false, 并中断 bpThread 线程的执行
    shouldServiceRun = false;
    if (bpThread != null) {
        bpThread.interrupt();
    }
}

// join() 方法只能由 blockPoolManager 调用
```

```
void join() {  
    // 直接调用 bpThread 的 join() 方法  
    try {  
        if (bpThread != null) {  
            bpThread.join();  
        }  
    } catch (InterruptedException ie) { }  
}
```

3. 工作线程 run()方法

BPServiceActor 的核心处理逻辑都是在 bpThread 工作线程中实现的，也就是在 BPServiceActor.run()方法中，本节我们学习 BPServiceActor.run()方法的实现。

BPServiceActor.run()方法会首先调用 BPServiceActor.connectToNNAndHandshake()方法与 Namenode 握手并初始化 Datanode 上该命名空间对应块池（BlockPool）的存储，然后在该 Namenode 上注册当前 Datanode。

执行完初始化操作之后，run()方法会循环调用 BPServiceActor.offerService()方法定期向 Namenode 发送心跳、块汇报、缓存汇报以及增量汇报。offerService()方法还会处理 Namenode 通过响应带回的名字节点指令。这里要注意，offerService()方法会一直循环执行，无论是否抛出异常，直到 shouldRunService 字段被赋值为 false 时才会停止执行。shouldRunService 字段在两种情况下会被赋值为 false：当 Datanode 关闭时以及客户端触发刷新 Namenode 操作时（Namenode 列表的改变，需要在当前命名空间中添加或者删除 Namenode）。

run()方法同时还维护了 BPServiceActor 的状态机。BPServiceActor 的初始状态为 CONNECTING，当 BPServiceActor 成功与 Namenode 连接并握手完成后，状态会转变为 RUNNING。如果连接和握手过程当中出现异常，并且有重试机制时，BPServiceActor 转变为 INIT_FAILED 状态。当调用 stop()方法将 shouldServiceRun 标志位设置为 false 后，BPServiceActor 更改为 EXITED 状态。BPServiceActor.run()方法的代码如下：

```
public void run() {  
    try {  
        while (true) {  
            // 初始化  
            try {  
                // 与 Namenode 握手，初始化 Datanode 存储并注册  
                connectToNNAndHandshake();  
                break;  
            } catch (IOException ioe) {  
                // 握手、存储恢复或者注册操作出现异常，将运行状态转换为 INIT_FAILED  
                runningState = RunningState.INIT_FAILED;  
                if (shouldRetryInit()) {  
                    // 进行重试操作，直到 BPOfferService 停止初始化操作  
                    sleepAndLogInterrupts(5000, "initializing");  
                } else {  
                    // 初始化失败，将运行状态改为 FAILED  
                }  
            }  
        }  
    }  
}
```



```

        runningState = RunningState.FAILED;
        return;
    }
}
// 初始化成功，将运行状态设置为 RUNNING
runningState = RunningState.RUNNING;

// 线程的主循环，循环调用 offerService() 方法向 Namenode 发送心跳、块汇报、增量块汇报以及缓存
// 块汇报，并执行 Namenode 返回的名字节点指令
while (shouldRun()) {
    try {
        offerService();
    } catch (Exception ex) {
        // 收到异常也不理会，继续循环直到 BPServiceActor 停止或者 Datanode 停止
        sleepAndLogInterrupts(5000, "offering service");
    }
}
// BPServiceActor 停止以后，将线程状态改为 EXITED
runningState = RunningState.EXITED;
} catch (Throwable ex) {
    // 如果出现异常，则将状态改为 FAILED
    runningState = RunningState.FAILED;
} finally {
    // 调用 cleanUp() 方法进行清理操作
    cleanUp();
}
}

```

下面我们看一下 `run()` 方法中调用的 `connectToNNAndHandshake()` 方法以及 `offerService()` 方法的实现。

（1）connectToNNAndHandshake()

`connectToNNAndHandshake()` 方法用于与 Namenode 握手，初始化 Datanode 存储并注册当前 Datanode。如下代码所示，`connectToNNAndHandshake()` 方法的操作分为以下 4 个步骤。

```

private void connectToNNAndHandshake() throws IOException {
    bpNamenode = dn.connectToNN(nnAddr);
    NamespaceInfo nsInfo = retrieveNamespaceInfo();
    bpos.verifyAndSetNamespaceInfo(nsInfo);
    register();
}

```

- 获取 Namenode 的 RPC 代理。
- 第一阶段的握手，通过调用 RPC 方法 `DatanodeProtocol.versionRequest()`，获取当前块池对应的命名空间的信息。
- 确认获取的命名空间信息与命名空间中另一个 Namenode 获取的信息匹配。同时，如果这是第一个 Namenode 的连接，则在 Datanode 上初始化命名空间对应块池的存储。

- 第二阶段的握手, 调用 `BPSERVICEActor.register()` 方法向 `Namenode` 注册当前 `Datanode`。

`BPSERVICEActor.connectToNNAndHandshake()` 方法的流程如图 4-31 所示, 我们结合流程图依次看一下这 4 个步骤的具体实现。

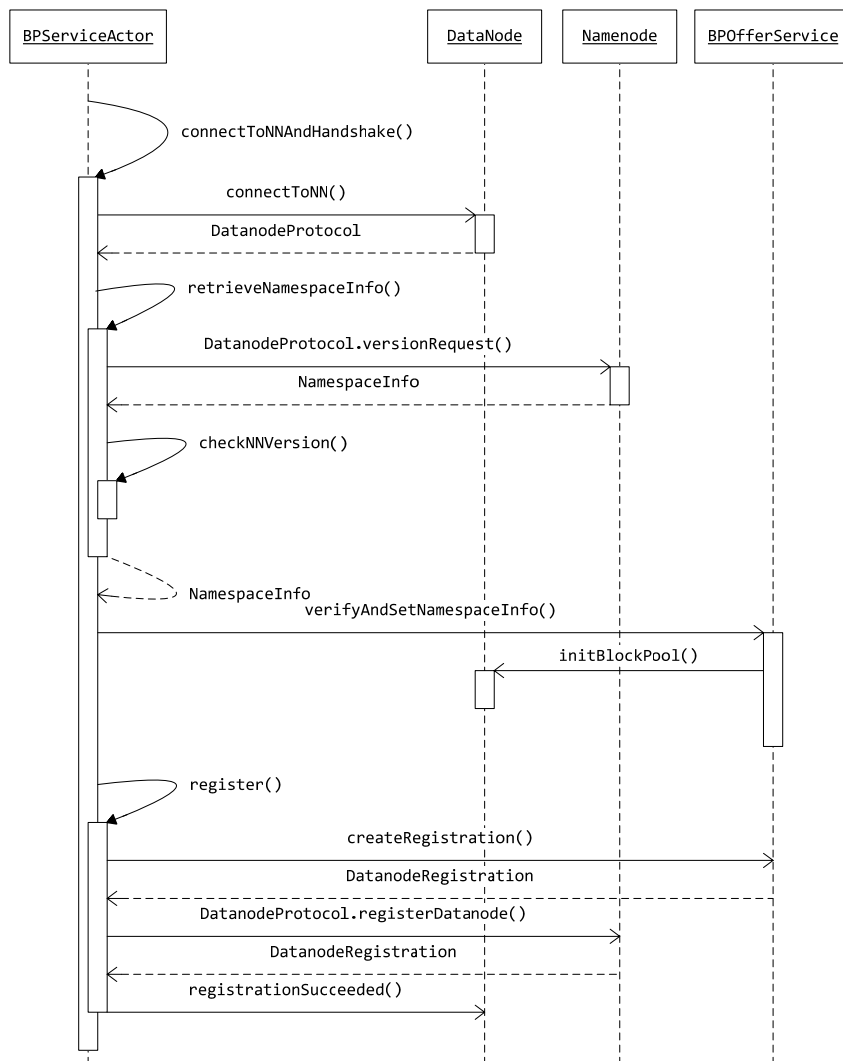


图 4-31 `connectToNNAndHandshake()` 方法流程图

`Datanode.connectToNN()`

`connectToNN()` 的实现非常简单, 就是构造 `DatanodeProtocolClientSideTranslatorPB` 引用, 作为 `Datanode` 向 `Namenode` 发起 RPC 调用的句柄。

retrieveNamespaceInfo()

调用 RPC 方法 `DatanodeProtocol.versionRequest()` 获取命名空间的信息，然后调用 `checkNNVersion()` 方法验证 Namenode 版本与当前 Datanode 版本的兼容性。

BPOfferService.verifyAndSetNamespaceInfo()

注意这个方法是在 `BPOfferService` 上调用的，`BPOfferService` 通过 `bpNSInfo` 字段保存当前块池对应的命名空间信息。如果这个字段为空，则证明当前 `BPSERVICEActor` 对应的 Namenode 是第一个响应握手请求的 Namenode。`verifyAndSetNamespaceInfo()` 方法会首先对 `bpNSInfo` 赋值，然后调用 `Datanode.initBlockPool()` 初始化命名空间对应块池的本地存储。如果这个字段不为空，则证明命名空间中定义的另一个 Namenode 已经提前注册并且初始化了本地存储，这时只需要将当前注册信息与已有的注册信息比较，确认 `blockPoolId`、`namespaceId`、`clusterId` 等字段相等即可。`BPOfferService.verifyAndSetNamespaceInfo()` 方法的代码如下：

```
void verifyAndSetNamespaceInfo(NamespaceInfo nsInfo) throws IOException {
    writeLock();
    try {
        if (this.bpNSInfo == null) {
            this.bpNSInfo = nsInfo;
            boolean success = false;
            // 第一个Namenode的响应，这时我们已经知道命名空间id，就可以通过调用
            // Datanode.initBlockPool()方法初始化Datanode的本地存储了
            try {
                dn.initBlockPool(this);
                success = true;
            } finally {
                if (!success) {
                    // 如果初始化失败，则将bpNSInfo置为空，等待下一个Namenode的响应
                    this.bpNSInfo = null;
                }
            }
        } else {
            // 检查blockPoolId、namespaceId、clusterId等字段的合法性
            checkNSEquality(bpNSInfo.getBlockPoolID(), nsInfo.getBlockPoolID(),
                "Blockpool ID");
            checkNSEquality(bpNSInfo.getNamespaceID(), nsInfo.getNamespaceID(),
                "Namespace ID");
            checkNSEquality(bpNSInfo.getClusterID(), nsInfo.getClusterID(),
                "Cluster ID");
        }
    } finally {
        writeUnlock();
    }
}
```

这里的 `Datanode.initBlockPool()` 操作初始化了命名空间对应块池的本地存储，在检查 `clusterId` 的一致性之后，会在 `BlockPoolManager` 中注册当前块池对应的 `BPOfferService` 对象。

initBlockPool()方法还会初始化块池对应的 DataStorage 对象（请参考 Datanode 存储的 DataStorage 小节），初始化 FsDatasetImpl 对象，以及 BlockScanner 和 DirectoryScanner 对象。

register()

第二阶段的握手，调用 DatanodeProtocol.registerDatanode()方法在 Namenode 上注册当前的 Datanode。注册成功后调用 BPOfferService.registrationSucceeded()方法确认 namespaceId、clusterId 与命名空间中定义的另一个 Namenode 返回的信息一致，同时确认 DatanodeUuid 与 Datanode 本地存储一致。BPOfferService.registrationSucceeded()方法的实现请参考 BPOfferService 的方法小节。register()方法的代码如下：

```
void register() throws IOException {
    // 构造 Datanode 注册请求，这里 StorageInfo 以及 DatanodeUuid 已经通过上一阶段的握手获得
    // 并且持久化在磁盘上，createRegistration() 根据这些信息构造 Datanode 注册请求
    bpRegistration = bpos.createRegistration();
    while (shouldRun()) {
        try {
            // 调用 DatanodeProtocol.registerDatanode() 注册 Datanode
            bpRegistration = bpNamenode.registerDatanode(bpRegistration);
            break;
        } catch (SocketTimeoutException e) {
            // Namenode 忙碌，则等待 1 秒后重试
            sleepAndLogInterrupts(1000, "connecting to server");
        }
    }
    // 调用 registrationSucceeded() 方法确认 namespaceId、clusterId
    // 与其他 Namenode 返回的信息一致
    // 并且确认 DatanodeUuid 与 Datanode 本地存储一致
    bpos.registrationSucceeded(this, bpRegistration);

    // 对块汇报做一个延迟
    scheduleBlockReport(dnConf.initialBlockReportDelay);
}
```

(2) offerService()

offerService()方法是 BPServiceActor 的主循环方法，它用于向 Namenode 发送心跳、块汇报、缓存汇报以及增量汇报。offerService()方法会一直循环运行，直到 Datanode 关闭或者客户端调用 ClientDatanodeProtocol.refreshNodes()重新加载 Namenode 配置。

offerService()的方法比较长，执行流程可参考图 4-32，我们同样将这个方法拆分成几个部分来看。

定期发送心跳

offerService()方法会以 dnConf.heartBeatInterval（默认配置是 3 秒）间隔向 Namenode 发送心跳。

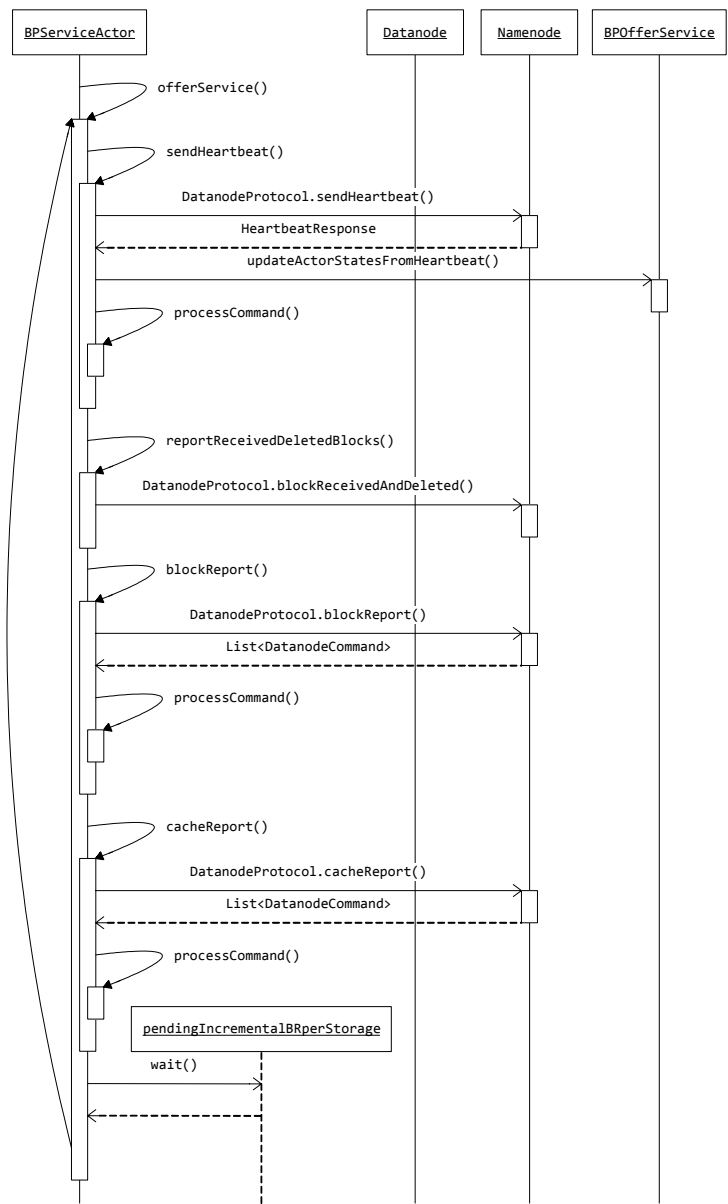


图 4-32 offerService()方法执行流程图

offerService()是通过调用 DatanodeProtocol.sendHeartbeat()方法向 Namenode 发送心跳的。Datanode 向 Namenode 发送的心跳信息主要包括: Datanode 注册信息、Datanode 存储信息(使用容量, 剩余容量等)、缓存信息、当前 Datanode 写文件的连接数, 以及读写数据使用的线程数等。这些数据描述了 Datanode 当前的负载情况。DatanodeProtocol.sendHeartbeat()方法的

定义如下代码所示。

```
public HeartbeatResponse sendHeartbeat(DatanodeRegistration registration,
                                       StorageReport[] reports,
                                       long dnCacheCapacity,
                                       long dnCacheUsed,
                                       int xmitsInProgress,
                                       int xceiverCount,
                                       int failedVolumes) throws IOException;
```

Namenode 收到 Datanode 的心跳之后，会返回一个心跳响应 HeartbeatResponse。这个心跳响应中包含一个 DatanodeCommand 的数组，用来携带 Namenode 对 Datanode 的名字节点指令。同时心跳响应中还包含一个 NNHASStatusHeartbeat 对象，用来标识当前 Namenode 的 HA 状态。Datanode 会使用这个字段来确定 BPOfferService 当中的哪一个 BPServiceActor 对应的 Namenode 是 Active 状态的。HeartbeatResponse 的定义如下代码所示。

```
public class HeartbeatResponse {
    private final DatanodeCommand[] commands;
    private final NNHASStatusHeartbeat haStatus;
    private final RollingUpgradeStatus rollingUpdateStatus;
```

下面我们看一下 offerService()方法中发送心跳流程部分的代码。offerService()方法首先调用 DatanodeProtocol.sendHeartbeat() 方法向 Namenode 发送一个心跳。然后调用 bpos.updateActorStatesFromHeartbeat() 方法处理心跳响应当中的 HA 状态字段 NNHASStatusHeartbeat，也就是更新 BPOfferService 中的 Active Namenode 的引用。最后调用 processCommand()方法执行 Namenode 通过心跳返回的名字节点指令。

```
// 每隔 dnConf.heartBeatInterval 进行一次心跳发送操作
if (startTime - lastHeartbeat >= dnConf.heartBeatInterval) {
    lastHeartbeat = startTime;
    if (!dn.areHeartbeatsDisabledForTests()) {
        // 调用 sendHeartbeat() 方法发送心跳至 Namenode
        HeartbeatResponse resp = sendHeartBeat();
        assert resp != null;
        dn.getMetrics().addHeartbeat(now() - startTime);

        // 对心跳响应中携带的 Namenode 的 HA 状态进行处理
        bpos.updateActorStatesFromHeartbeat(
            this, resp.getNameNodeHaState());
        state = resp.getNameNodeHaState().getState();

        if (state == HAServiceState.ACTIVE) {
            handleRollingUpgradeStatus(resp);
        }
        // 最后调用 processCommand() 方法处理响应中带回的 Namenode 指令
        long startProcessCommands = now();
        if (!processCommand(resp.getCommands()))
            continue;
```

```

        long endProcessCommands = now();
    }
}

```

BPOfferService.updateActorStatesFromHeartbeat() 方法用于处理心跳响应中携带的 Namenode HA 状态。例如原来的 Standby Namenode 切换成 Active Namenode。如果发生这种情况，就需要对 BPOfferService 的 bpServiceToActive 字段进行更改（bpServiceToActive 字段记录了 BPOfferService 认为是 Active 状态的 Namenode 对应的 BPServiceActor 对象的引用）。为了防止脑裂，也就是两个 Namenode 都声明自己是 Active Namenode 的情况。Namenode 心跳响应中的 NNHASStatusHeartbeat 对象专门携带了一个 txid 字段，只有携带大的 txid 的 Namenode 才能作为 Active Namenode。另一种情况是原先 Active 状态的 Namenode 切换为 Standby 状态，这时直接将 BPOfferService.bpServiceToActive 字段赋值为 null 即可。这里我们看一下 BPOfferService.updateActorStatesFromHeartbeat() 的代码。

```

void updateActorStatesFromHeartbeat(
    BPServiceActor actor,
    NNHASStatusHeartbeat nnHaState) {
    writeLock();
    try {
        // Namenode 携带的 txid
        final long txid = nnHaState.getTxId();
        // 当前 Namenode 是否声明自己为 Active Namenode
        final boolean nnClaimsActive =
            nnHaState.getState() == HAServiceState.ACTIVE;
        // BPOfferService 是否认为当前 Namenode 为 Active Namenode
        final boolean bposThinksActive = bpServiceToActive == actor;
        // 当前 Namenode 携带的 txid 是否大于原 Active Namenode 携带的 txid
        final boolean isMoreRecentClaim = txid > lastActiveClaimTxId;

        // 原来的 Standby Namenode 声明自己为 Active Namenode, 发生状态切换
        if (nnClaimsActive && !bposThinksActive) {
            // 当前 Namenode 携带的 txid 小于原 Active Namenode 的 txid, 也就是有两个
            // Namenode 声明自己为 Active, 但是当前 Namenode 的请求过时
            if (!isMoreRecentClaim) {
                // 对于过时的请求, 则直接忽略
                return;
            } else { // 当前的请求是最新的请求
                if (bpServiceToActive == null) {
                    // BPOfferService 还没有保存 active namenode
                } else {
                    // 当前 Namenode 的请求是最新的请求
                }
                // 将 bpServiceToActive 指向当前 Namenode 对应的 BPServiceActor
                bpServiceToActive = actor;
            }
        } else if (!nnClaimsActive && bposThinksActive) {
            // 原来 Active 状态的 Namenode 现在声明自己为 Standby

```

```
// 则直接将 bpServiceToActive 赋值为 null
bpServiceToActive = null;
}

// 更新 lastActiveClaimTxid
if (bpServiceToActive == actor) {
    assert txid >= lastActiveClaimTxId;
    lastActiveClaimTxId = txid;
}
} finally {
    writeUnlock();
}
}
```

完成上面的操作以后，offerService()方法就会调用 processCommand()方法执行 Namenode 返回的名字节点指令。processCommand()方法首先遍历心跳响应中携带的指令，然后调用 BPOfferService.processCommandFromActor()方法处理指令。这个方法我们在 BPOfferService 实现的响应名字节点指令小节中学习。

最近新添加和删除的数据块

Datanode 除了定期向 Namenode 发送心跳外，还需要向 Namenode 汇报 Datanode 最近新添加和删除的数据块（汇报间隔是 100 *心跳间隔，也就是 300 秒）。在 offerService()方法中，调用 reportReceivedDeletedBlocks()方法执行这个操作。我们看一下 offerService()中的相关代码。

```
if (sendImmediateIBR ||
    (startTime - lastDeletedReport > dnConf.deleteReportInterval)) {
    // 定时调用 reportReceivedDeletedBlocks() 发送增量块汇报
    reportReceivedDeletedBlocks();
    lastDeletedReport = startTime;
}
// ...
long waitTime = dnConf.heartBeatInterval -
    (Time.now() - lastHeartbeat);
synchronized(pendingIncrementalBRperStorage) {
    if (waitTime > 0 && !sendImmediateIBR) {
        try {
            // 在 pendingIncrementalBRperStorage 上等待
            pendingIncrementalBRperStorage.wait(waitTime);
        } catch (InterruptedException ie) {
            LOG.warn("BPOfferService for " + this + " interrupted");
        }
    }
}
```

这里我们先看一下 BPSERVICEActor.pendingIncrementalBRperStorage 这个数据结构的维护。pendingIncrementalBRperStorage 是一个 Map，维护 DataStorage 上存储的两次汇报之间新添加和删除的数据块。BPSERVICEActor 提供了 notifyNamenodeBlock()以及 notifyNamenodeDeletedBlock()

方法用于向 `pendingIncrementalBRperStorage` 中添加更新的数据块信息。`notifyNamenodeBlock()` 方法用于添加一个 `Datanode` 新接收的数据块信息, 而 `notifyNamenodeDeletedBlock()` 方法则用于添加一个 `Datanode` 新删除的数据块信息。这里要注意的是, 对于新添加的数据块, `notifyNamenodeBlock()` 方法会将 `sendImmediateIBR` 字段设置为 `true`, 并且唤醒在 `pendingIncrementalBRperStorage` 上等待新数据块的 `offerService()` 方法, 也就是 `offerService()` 会立即将添加数据块的信息发送给 `Namenode`。这么做的原因是 `Datanode` 接收到新的数据块后, 期望立即发送响应给客户端, 而不是阻塞在 `Namenode` 的通知上。这里我们看一下 `BPSERVICEActor` 中与维护 `pendingIncrementalBRperStorage` 字段相关的方法。

```
// 维护一个 DataStorage 上新添加的数据块信息以及刚删除的数据块信息
private final Map<DatanodeStorage, PerStoragePendingIncrementalBR>
    pendingIncrementalBRperStorage = Maps.newHashMap();

// 将新添加的数据块信息加入 pendingIncrementalBRperStorage 中
void notifyNamenodeBlock(ReceivedDeletedBlockInfo bInfo,
    String storageUuid, boolean now) {
    synchronized (pendingIncrementalBRperStorage) {
        addPendingReplicationBlockInfo(
            bInfo, dn.getFSDataset().getStorage(storageUuid));
        sendImmediateIBR = true;
        // 立即唤醒 offerService() 方法, 将新添加的数据块信息汇报给 Namenode
        if (now) {
            pendingIncrementalBRperStorage.notifyAll();
        }
    }
}

// 将刚删除的数据块信息加入 pendingIncrementalBRperStorage 中
void notifyNamenodeDeletedBlock(
    ReceivedDeletedBlockInfo bInfo, String storageUuid) {
    synchronized (pendingIncrementalBRperStorage) {
        addPendingReplicationBlockInfo(
            bInfo, dn.getFSDataset().getStorage(storageUuid));
    }
}
```

`BPSERVICEActor.notifyNamenodeBlock()` 和 `notifyNamenodeDeletedBlock()` 方法是在 `BPOfferService.notifyNamenodeDeletedBlock()`、`BPOfferService.notifyNamenodeReceivedBlock()`、`BPOfferService.notifyNamenodeReceivingBlock()` 中调用的。这三个方法我们在 `BPOfferService` 实现小节中再详细介绍。

了解了 `pendingIncrementalBRperStorage` 字段, 我们再来看 `reportReceivedDeletedBlocks()` 方法的实现。`reportReceivedDeletedBlocks()` 方法会首先从 `pendingIncrementalBRperStorage` 对象中取出两次块汇报之间增量的、还没有向 `Namenode` 汇报的数据块信息。然后将这些信息放入 `reports` 变量中, 并通过调用 RPC 方法 `DatanodeProtocol.blockReceivedAndDeleted()` 将汇报发送给 `Namenode`。如果这次 RPC 调用失败, 则将 `reports` 变量中的信息重新放回

pendingIncrementalBRperStorage 中。然后将 sendImmediateIBR 赋值为 true，立即重发 ReceivedDeletedBlockReport。如果当前没有工作且 sendImmediateIBR 为 false，那么 offerService()方法会在 pendingIncrementalBRperStorage 对象上等待，直到超时或者被唤醒。reportReceivedDeletedBlocks()方法的代码实现如上描述，由于篇幅原因这里就不再贴出代码了。

数据块汇报

Datanode 会在启动时或者间隔 blockReportInterval（默认是 6 个小时）时向 Namenode 发送块汇报，块汇报包括 Datanode 上存储的所有数据块信息。Namenode 会通过响应返回名字节点指令。我们看一下 offerService()中发送块汇报部分的代码。offerService()首先调用 blockReport()方法发送块汇报，然后接收 Namenode 返回的名字节点指令 cmds，最后调用 processCommand()方法执行这些指令。

```
List<DatanodeCommand> cmds = blockReport();  
processCommand(cmds == null ? null : cmds.toArray(new DatanodeCommand[cmds.size()]));
```

blockReport()方法每隔 blockReportInterval（默认间隔是 6 个小时）会触发一次块汇报。它首先调用 DatanodeProtocol.reportReceivedDeletedBlocks()方法向 Namenode 汇报 Datanode 最近添加与删除的数据块，以避免 Namenode 和 Datanode 元数据不同步。然后调用 FSDatasetImpl.getBlockReports()获得当前块池的块汇报信息（请参考文件系统数据集的 FSDatasetImpl 小节）。接下来将获取的块汇报信息转换为 StorageBlockReport 类型，再通过 RPC 方法 DatanodeProtocol.blockReport()发送这个数据块汇报到 Namenode。最后保存 Namenode 返回的名字节点指令。blockReport()方法的实现完全按照以上步骤，比较简单，这里就不贴出代码了。

processCommand()方法我们在发送心跳部分已经介绍了，这里不再赘述。

缓存数据块汇报

缓存汇报的逻辑类似于块汇报，offerService()方法首先调用 cacheReport()方法，向 Namenode 汇报当前 Datanode 的缓存情况，然后调用 processCommand()方法执行 Namenode 携带回的指令。

```
DatanodeCommand cmd = cacheReport();  
processCommand(new DatanodeCommand[]{ cmd });
```

cacheReport()方法会每隔 cacheReportInterval（默认间隔是 10 秒）进行一次块汇报。这里首先会调用 FsDatasetImpl.getCacheReport()方法获取所有缓存的数据块（请参考文件系统数据集的 FSDatasetImpl 小节），然后调用 RPC 方法 ClientProtocol.cacheReport()将所有缓存的数据块汇报给 Namenode，最后保存 Namenode 返回的名字节点指令。

启动数据块扫描操作

完成上述操作之后，offerService()方法会启动当前块池的数据块扫描功能（请参考数据块扫描器小节）。

```

if (dn.blockScanner != null) {
    dn.blockScanner.addBlockPool(bpos.getBlockPoolId());
}

```

线程睡眠等待

完成上述操作之后，`offerService()`方法会在 `pendingIncrementalBRperStorage` 对象上等待，直到下一个心跳周期或者被唤醒。这里的唤醒操作是在 `notifyNamenodeBlock()`方法中执行的，我们已经在最近新添加和删除的数据块小节中介绍过了，此处不再赘述。

```

long waitTime = dnConf.heartBeatInterval -
    (Time.now() - lastHeartbeat);
synchronized(pendingIncrementalBRperStorage) {
    if (waitTime > 0 && !sendImmediateIBR) {
        try {
            pendingIncrementalBRperStorage.wait(waitTime);
        } catch (InterruptedException ie) {
            LOG.warn("BPOfferService for " + this + " interrupted");
        }
    }
}

```

`offerService()`方法对于发生 Datanode 注册信息错误、Datanode 不合法（不在 `include` 列表中）或者 `VERSION` 信息错误等异常情况，会直接关闭当前 `BPSERVICEActor` 并停止线程运行。对于其他异常，`offerService()`方法都会忽略，继续循环执行逻辑。

至此，`BPSERVICEActor` 的实现我们就介绍完了。下一节我们介绍 `BPOfferService` 的实现。

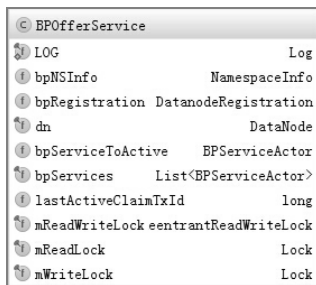
4.4.2 BPOfferService 实现

通过前面的介绍我们知道，在 HDFS Federation 部署中，一个 HDFS 集群可以定义多个命名空间，每一个命名空间在 Datanode 上都有一个对应的块池存储这个命名空间的数据块，这个块池是由一个 `BPOfferService` 实例管理的。由于一个命名空间可以定义两个 Namenode，所以 `BPOfferService` 类需要与两个 Namenode 通信并执行指定的逻辑，也就是 `BPOfferService` 要拥有两个 `BPSERVICEActor` 对象的引用。同时为了防止出现脑裂的情况，需要保证命名空间中有且只能有一个处于活动状态的 Namenode，`BPOfferService` 类还需要管理当前 Datanode 认为是 Active 状态的 Namenode 的引用（通过 `bpServiceToActive` 字段）。

1. BPOfferService 的字段 (done)

我们首先来介绍 `BPOfferService` 的字段。如图 4-33 所示，`BPOfferService` 定义了如下字段。

- **NamespaceInfo bpNSInfo:** 当前 `BPOfferService` 服务的命名空间的信息，这个信息是在与 Namenode 的握手阶段获得的。
- **DatanodeRegistration bpRegistration:** 当前 `BPOfferService` 对应的块池在 Namenode 上的注册信息，这个信息是在 Datanode 注册阶段获得。



LOG	Log
bpNSInfo	NamespaceInfo
bpRegistration	DatanodeRegistration
dn	DataNode
bpServiceToActive	BPSERVICEActor
bpServices	List<BPSERVICEActor>
lastActiveClaimTxId	long
mReadWriteLock	entrantReadWriteLock
mReadLock	Lock
mWriteLock	Lock

图 4-33 BPOfferService 定义的字段

- DataNode dn: 当前 DataNode 对象的引用。
- BPSERVICEActor bpServiceToActive: 当前 BPOfferService 认为 Active 的 Namenode 对应的 BPSERVICEActor 对象。
- List<BPSERVICEActor> bpServices: 当前命名空间中所有 Namenode 对应的 BPSERVICEActor 的列表。注意这里是一个 CopyOnWriteList。
- lastActiveClaimTxId: 每当收到一个 Namenode(声明自己为 Active 状态的 Namenode) 传来的心跳时, 就记录下最近的一个 transactionId, 这个字段用于防止出现脑裂的情况。

2. BPOfferService 的方法

BPOfferService 定义的方法比较多, 我们分为以下几类来学习。

(1) 触发汇报

trySendErrorReport()、reportRemoteBadBlock()、reportBadBlocks()方法实现了向 Namenode 发送错误汇报、汇报远程坏块以及本地坏块的操作, 这三个方法会直接调用两个 BPSERVICEActor 对象的对应方法, BPSERVICEActor 对象则会调用 DatanodeProtocol 对应的方法执行汇报操作。下面我们以 reportBadBlocks()方法为例, reportBadBlocks()方法调用了当前 BPOfferService 持有的所有 BPSERVICEActor 对象的 reportBadBlocks()方法, 向 Namenode 汇报损坏的数据块。

```
void reportBadBlocks(ExtendedBlock block,
                    String storageUuid, StorageType storageType) {
    checkBlock(block);
    // 遍历 BPOfferService 中管理的所有 BPSERVICEActor 对象
    for (BPSERVICEActor actor : bpServices) {
        // 在这些对象上调用对应的 reportBadBlocks()方法
        actor.reportBadBlocks(block, storageUuid, storageType);
    }
}
```

BPSERVICEActor.reportBadBlocks()又会调用 RPC 方法 ClientProtocol.reportBadBlocks()向该 BPSERVICEActor 对应的 Namenode 汇报错误的数据块。代码如下:

```
void reportBadBlocks(ExtendedBlock block,
```

```

String storageUuid, StorageType storageType) {
    if (bpRegistration == null) {
        return;
    }
    DatanodeInfo[] dnArr = { new DatanodeInfo(bpRegistration) };
    String[] uuids = { storageUuid };
    StorageType[] types = { storageType };
    LocatedBlock[] blocks = { new LocatedBlock(block, dnArr, uuids, types) };

    try {
        // 调用 ClientProtocol.reportBadBlocks() 向对应的 Namenode 汇报错误的数据块
        bpNamenode.reportBadBlocks(blocks);
    } catch (IOException e) {
    }
}
}

```

trySendErrorReport()、reportRemoteBadBlock()方法也是同样的逻辑,感兴趣的读者请自行阅读代码。

(2) 添加与删除数据块操作

如图 4-34 所示,当 Datanode 接收一个新的数据块时,如客户端通过数据流管道写入一个数据块、块恢复操作更新了 RUR 状态数据块的时间戳和文件长度时,或者通过 DataTransfer Protocol 流式接口复制一个数据块时,都会调用 BPOfferService.notifyNamenodeReceivedBlock()方法通知命名空间。

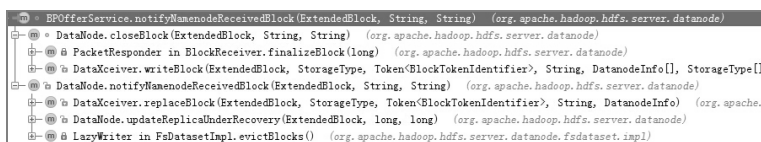


图 4-34 notifyNamenodeReceivedBlock()调用关系

当 Datanode 删除一个已有的数据块时,会调用 BPOfferService.notifyNamenodeDeletedBlock()方法通知命名空间。如图 4-35 所示,当 Datanode 调用 FsDatasetImpl.invalidate()方法从 Datanode 上删除一个数据块时,invalidate()方法会创建一个删除任务 ReplicaFileDeleteTask 异步地从 Datanode 磁盘上删除这个数据块文件,当删除操作完成后会调用 BPOfferService.notifyNamenodeDeletedBlock()方法通知命名空间。

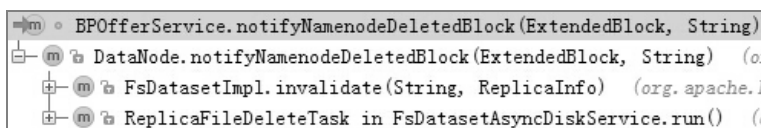


图 4-35 notifyNamenodeDeletedBlock()调用关系

notifyNamenodeDeletedBlock、notifyNamenodeReceivingBlock、notifyNamenodeReceivedBlock 这三个方法的实现都是遍历 BPOfferService 持有的所有 BPServiceActor 对象,并在

BPSERVICEActor 对象上调用 `notifyNamenodeBlockImmediately()`、`notifyNamenodeDeletedBlock()` 方法。这里我们以 `notifyNamenodeDeletedBlock()` 为例：

```
void notifyNamenodeDeletedBlock(ExtendedBlock block, String storageUuid) {
    checkBlock(block);
    ReceivedDeletedBlockInfo bInfo = new ReceivedDeletedBlockInfo(
        block.getLocalBlock(), BlockStatus.DELETED_BLOCK, null);
    // 遍历所有的 BPSERVICEActor 对象，并调用 notifyNamenodeDeletedBlock() 方法
    for (BPSERVICEActor actor : bpServices) {
        actor.notifyNamenodeDeletedBlock(bInfo, storageUuid);
    }
}
```

`BPSERVICEActor.notifyNamenodeDeletedBlock()` 方法的实现请参考 `BPSERVICEActor` 小节 的分析。

(3) 管理 `bpNSInfo`、`bpRegistration`、`bpServiceToActive`

在 `BPOfferService` 中维护着三个比较重要的字段。

NamespaceInfo bpNSInfo

`bpNSInfo` 字段保存了当前 `BPOfferService` 服务块池的命名空间信息，这个字段是在 `Datanode` 与 `Namenode` 握手时调用 `BPOfferService.verifyAndSetNamespaceInfo()` 方法设置的（请参考 `BPSERVICEActor` 实现小节）。`verifyAndSetNamespaceInfo()` 方法还会校验当前命名空间中两个 `Namenode` 获取的命名空间信息（`NamespaceInfo`）是否一致。我们看一下 `verifyAndSetNamespaceInfo()` 方法的实现。

```
void verifyAndSetNamespaceInfo(NamespaceInfo nsInfo) throws IOException {
    writeLock();
    try {
        if (this.bpNSInfo == null) {
            this.bpNSInfo = nsInfo;
            boolean success = false;
            // 第一个 Namenode 的响应，这时我们已经知道命名空间 id，就可以通过 Datanode 引用初始化 Datanode
            // 的本地存储了
            try {
                dn.initBlockPool(this);
                success = true;
            } finally {
                if (!success) {
                    // 如果初始化失败，则将 bpNSInfo 置为空，等待下一个 Namenode 的响应
                    this.bpNSInfo = null;
                }
            }
        } else {
            // 检查 blockPoolId、namespaceId、clusterId 等字段是否一致
            checkNSEquality(bpNSInfo.getBlockPoolID(), nsInfo.getBlockPoolID(),
                "Blockpool ID");
        }
    }
}
```

```

        checkNSEquality(bpNSInfo.getNamespaceID(), nsInfo.getNamespaceID(),
            "Namespace ID");
        checkNSEquality(bpNSInfo.getClusterID(), nsInfo.getClusterID(),
            "Cluster ID");
    }
} finally {
    writeUnlock();
}
}
}

```

DatanodeRegistration bpRegistration

bpRegistration 字段保存了当前 BPOfferService 管理的块池在 Namenode 上的注册信息。这个信息是在 Datanode 向 Namenode 注册的过程中, 通过调用 registrationSucceeded()方法设置的。registrationSucceeded() 方法还会验证从命名空间中两个 Namenode 获取的 DatanodeRegistration 是否一致。

```

void registrationSucceeded(BPServiceActor bpServiceActor,
    DatanodeRegistration reg) throws IOException {
    writeLock();
    try {
        if (bpRegistration != null) {
            // 验证已经保存的注册信息与当前注册信息是否一致
            checkNSEquality(bpRegistration.getStorageInfo().getNamespaceID(),
                reg.getStorageInfo().getNamespaceID(), "namespace ID");
            checkNSEquality(bpRegistration.getStorageInfo().getClusterID(),
                reg.getStorageInfo().getClusterID(), "cluster ID");
        } else {
            // bpRegistration 还没有初始化, 则用当前注册信息初始化 bpRegistration
            bpRegistration = reg;
        }

        dn.bpRegistrationSucceeded(bpRegistration, getBlockPoolId());
    }
} finally {
    writeUnlock();
}
}

```

BPServiceActor bpServiceToActive

bpServiceToActive 字段保存了当前 BPOfferService 认为活动的 Namenode 对应的 BPServiceActor 对象, 这个字段是在 updateActorStatesFromHeartbeat()方法中赋值的。updateActorStatesFromHeartbeat()方法我们在 BPServiceActor 实现的 offerService()小节中已经介绍过了, 这里就不再介绍了。

(4) 响应名字节点指令

Datanode 会通过心跳、块汇报、缓存汇报的响应携带回名字节点指令, BPOfferService

提供了 `processCommandFromActor()` 方法处理名字节点指令。需要特别注意的是，对于 `ActiveNamenode` 和 `Standby Namenode` 返回的名字节点指令，处理逻辑是不同的。

`processCommandFromActor()` 方法是 `BPOfferService` 处理名字节点指令的入口方法，`BPSERVICEActor` 在工作线程的 `offerService()` 方法中接收到 `Namenode` 返回的指令后，会调用这个方法处理名字节点指令。`processCommandFromActor()` 方法的代码如下：

```
boolean processCommandFromActor(DatanodeCommand cmd,
    BPServiceActor actor) throws IOException {
    assert bpServices.contains(actor);
    if (cmd == null) {
        return true;
    }

    if (DatanodeProtocol.DNA_REGISTER == cmd.getAction()) {
        // 如果 Namenode 返回的指令要求 Datanode 重新注册，则调用 reRegister() 方法
        actor.reRegister();
        return false;
    }
    writeLock();
    try {
        // 对于 Active Namenode 返回的指令，调用 processCommandFromActive() 方法处理
        if (actor == bpServiceToActive) {
            return processCommandFromActive(cmd, actor);
        } else {
            // 对于 Standby Namenode 返回的指令，则调用 processCommandFromStandby() 方法处理
            return processCommandFromStandby(cmd, actor);
        }
    } finally {
        writeUnlock();
    }
}
```

可以看到，对于 `Active Namenode` 和 `Standby Namenode` 返回的指令，处理方法是完全不同的。下面我们分别看一下这两个方法的实现。

`processCommandFromStandby()` 类

`processCommandFromStandby()` 处理来自 `StandbyNamenode` 的名字节点指令。这个方法的处理逻辑很简单，对于 `Standby Namenode` 返回的指令，直接忽略即可。这样处理的原因是为了防止在 HA 部署下出现脑裂的情况，也就是 `Active Namenode` 和 `Standby Namenode` 同时向 `Datanode` 下发指令。所以 `BPOfferService` 对象并不执行 `Standby Namenode` 返回的名字节点指令，只执行 `Active Namenode` 返回的指令，这样也就保证了单个命名空间中只有一个 `Active` 状态的 `Namenode`。`processCommandFromStandby()` 方法的代码如下：

```
private boolean processCommandFromStandby(DatanodeCommand cmd,
    BPServiceActor actor) throws IOException {
    switch (cmd.getAction()) {
```



```
// 安全相关
// 忽略其他所有名字节点指令
case DatanodeProtocol.DNA_TRANSFER:
case DatanodeProtocol.DNA_INVALIDATE:
case DatanodeProtocol.DNA_SHUTDOWN:
case DatanodeProtocol.DNA_FINALIZE:
case DatanodeProtocol.DNA_RECOVERBLOCK:
case DatanodeProtocol.DNA_BALANCERBANDWIDTHUPDATE:
case DatanodeProtocol.DNA_CACHE:
case DatanodeProtocol.DNA_UNCACHE:
    break;
default:
}
return true;
}
```

processCommandFromActive()

processCommandFromActive()方法处理来自 Active Namenode 的名字节点指令，除了 REGISTER 指令已经在 processCommandFromActor()中处理了，其他事件的处理逻辑请参考图 4-36。可以看到，这里的处理逻辑都是比较简单的直接方法调用，此处就不再贴出代码了，请读者自己参考源码。

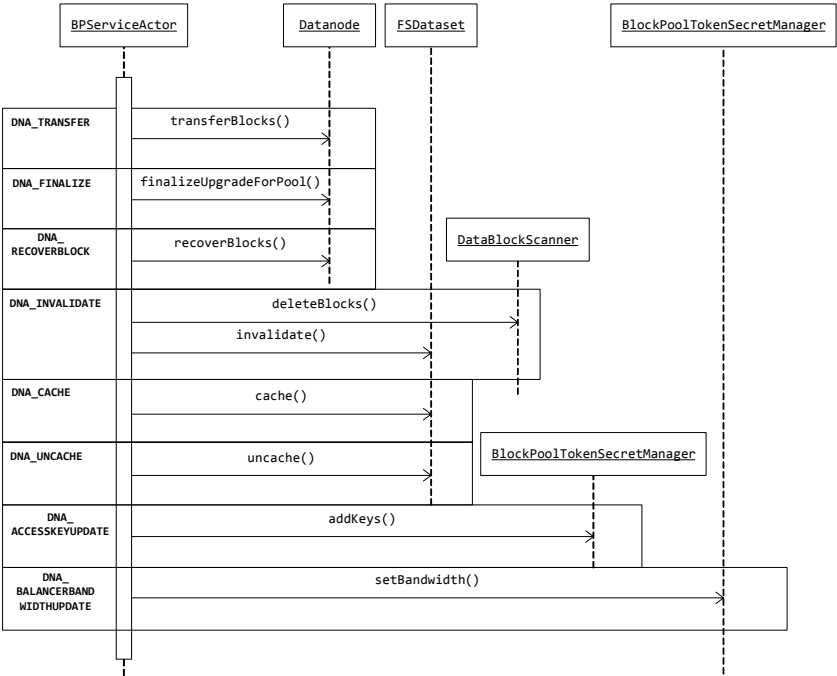


图 4-36 processCommandFromActive()方法处理逻辑图

4.4.3 BlockPoolManager 实现

BlockPoolManager 类负责管理所有的 BPOfferService 实例，对外提供添加、删除、启动、停止、关闭 BPOfferService 类的接口。所有对 BPOfferService 的操作，都必须通过 BlockPoolManager 类提供的方法来执行。

DataNode.startDataNode() 方法初始化了 BlockPoolManager 对象，然后调用 BlockPoolManager.refreshNamenodes()方法完成对 BlockPoolManager 的构造。这一节我们就重点讲解 BlockPoolManager 类以及入口方法 refreshNamenodes()的实现。

1. BlockPoolManager 的字段 (done)

BlockPoolManager 最重要的功能就是维护 Datanode 上所有 BPOfferService 对象的引用，同时对外提供多种检索 BPOfferService 的方式——通过命名空间 id (nameserviceId) 检索、通过块池 id (blockPoolId) 检索等。所以 BlockPoolManager 的字段主要是 BPOfferService 的映射以及集合。

```
private final Map<String, BPOfferService> bpByNameserviceId =
    Maps.newHashMap(); // 命名空间 id 与 BPOfferService 的映射
private final Map<String, BPOfferService> bpByBlockPoolId =
    Maps.newHashMap(); // 块池 id 与 BPOfferService 的映射
private final List<BPOfferService> offerServices =
    Lists.newArrayList();
```

2. refreshNamenodes()方法 (done)

refreshNamenodes() 方法用于根据 HDFS 配置添加、删除以及更新命名空间。在 BlockPoolManager 的实现中，就是对指定命名空间的 BPOfferService 引用进行更新。这里的操作分为如下几步。

(1) 构造 toRefresh、toAdd、toRemove 三个队列

添加、更新、删除 BPOfferService，方法中的 addrMap 变量保存了配置的命名空间列表，bpByNameserviceId 变量则保存了当前 BlockPoolManager 中已有的命名空间列表。

- 如果命名空间列表中存在，BlockPoolManager 的 Map 中不存在，则将当前的 nameserviceId 添加到 toAdd 队列中。
- 如果命名空间列表中存在，BlockPoolManager 的 Map 中也存在，则将当前的 nameserviceId 添加到 toRefresh 队列中。
- 如果 BlockPoolManager 的 Map 中存在，命名空间列表中不存在，则将当前的 nameserviceId 添加到 toRemove 队列中。

```
for (String nameserviceId : addrMap.keySet()) {
    if (bpByNameserviceId.containsKey(nameserviceId)) {
        toRefresh.add(nameserviceId);
    } else {
        toAdd.add(nameserviceId);
    }
}
```

```

    }
}

toRemove = Sets.newHashSet(Sets.difference(
    bpByNameserviceId.keySet(), addrMap.keySet()));

```

（2）处理 toAdd 队列

toAdd 队列中保存了需要添加到 BlockPoolManager 中的命名空间，refreshNamenodes()方法会首先调用 createBPOS()方法创建 BPOfferService 对象，BPOfferService 的构造方法会对命名空间中定义的每一个 Namenode 构造对应的 BPSERVICEActor 对象，并将这些新创建的对象放入 BPOfferService 的数据结构中保存。BPOfferService 构造完成后，将会放入 BlockPoolManager.bpByNameserviceId 映射中保存，然后调用 startAll()方法启动所有的 BPOfferService，BPOfferService 会级联启动所有的 BPSERVICEActor 的工作线程。

```

if (!toAdd.isEmpty()) {
    for (String nsToAdd : toAdd) {
        ArrayList<InetSocketAddress> addrs =
            Lists.newArrayList(addrMap.get(nsToAdd).values());
        // 调用 createBPOS()方法创建 BPOfferService 对象
        BPOfferService bpos = createBPOS(addrs);
        // 加入 bpByNameserviceId 映射中
        bpByNameserviceId.put(nsToAdd, bpos);
        offerServices.add(bpos);
    }
}
// 启动所有的 BPOfferService，级联启动 BPSERVICEActor 的工作线程
startAll();

```

（3）处理 toRemove 队列

toRemove 队列中保存了要从 BlockPoolManager 中删除的命名空间。对于 toRemove 队列中保存的命名空间，停止它对应的 BPOfferService 的服务，也就是停止 BPOfferService 中所有的 BPSERVICEActor 线程。当 BPOfferService 的所有 BPSERVICEActor 线程都停止后，会自动从 BlockPoolManager.bpByNameserviceId 映射中移除当前的 BPOfferService 对象。

```

if (!toRemove.isEmpty()) {
    for (String nsToRemove : toRemove) {
        BPOfferService bpos = bpByNameserviceId.get(nsToRemove);
        bpos.stop();
        bpos.join();
    }
}

```

（4）处理 toRefresh 队列

对于 toRefresh 队列中保存的命名空间，则调用 BPOfferService.refreshNNList()方法更新该命名空间中的 Namenode 列表。

```
if (!toRefresh.isEmpty()) {
    LOG.info("Refreshing list of NNs for nameservices: " +
        Joiner.on(",").useForNull("<default>").join(toRefresh));

    for (String nsToRefresh : toRefresh) {
        BPOfferService bpos = bpByNameserviceId.get(nsToRefresh);
        ArrayList<InetSocketAddress> addrs =
            Lists.newArrayList(addrMap.get(nsToRefresh).values());
        bpos.refreshNNList(addrs);
    }
}
```

这里要注意的是，HDFS 目前版本的 HA 并不支持在运行的 **Datanode** 上添加一个新的 **StandbyNamenode**，所以如果遇到这种情况，**BPOfferService.refreshNNList()**方法会直接抛出一个异常。可以说这个方法目前的实现是为后续版本做准备的，并没有执行实际的操作。

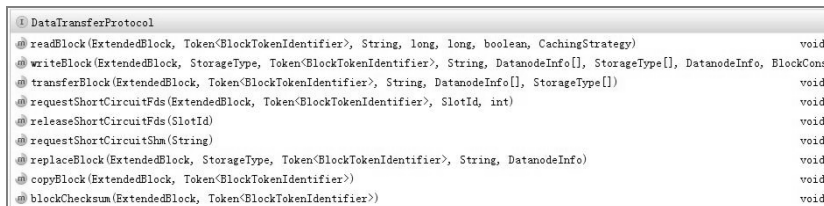
4.5 流式接口

数据节点最重要的功能就是管理物理存储上的数据块，并与 **Namenode** 以及 **DFSClient** 通信以执行读写数据块的操作。这里的读写操作涉及大量数据的传输，例如 **DFSClient** 将数据块写入数据节点中、**DFSClient** 从数据节点中读取数据块，以及数据节点将数据块复制到其他数据节点等，这些操作都涉及大量的 IO。

在 **Datanode** 的实现中，对这些读写操作提供了基于 **TCP** 流的数据访问接口 **DataTransferProtocol**。本节就介绍 **DataTransferProtocol** 的实现。

4.5.1 DataTransferProtocol 定义

DataTransferProtocol 是用来描述写入或者读出 **Datanode** 上数据的流式接口。如图 4-37 所示，**DataTransferProtocol** 定义了如下方法，其中最重要的方法就是 **readBlock()**和 **writeBlock()**。



DataTransferProtocol	
@readBlock(ExtendedBlock, Token<BlockTokenIdentifier>, String, long, long, boolean, CachingStrategy)	void
@writeBlock(ExtendedBlock, StorageType, Token<BlockTokenIdentifier>, String, DatanodeInfo[], StorageType[], DatanodeInfo, BlockCons	
@transferBlock(ExtendedBlock, Token<BlockTokenIdentifier>, String, DatanodeInfo[], StorageType[])	void
@requestShortCircuitFds(ExtendedBlock, Token<BlockTokenIdentifier>, SlotId, int)	void
@releaseShortCircuitFds(SlotId)	void
@requestShortCircuitShm(String)	void
@replaceBlock(ExtendedBlock, StorageType, Token<BlockTokenIdentifier>, String, DatanodeInfo)	void
@copyBlock(ExtendedBlock, Token<BlockTokenIdentifier>)	void
@blockChecksum(ExtendedBlock, Token<BlockTokenIdentifier>)	void

图 4-37 DataTransferProtocol 接口

- **readBlock()**: 从当前 **Datanode** 上读取指定的数据块。
- **writeBlock()**: 将当前 **Datanode** 上存储的指定数据块写入数据流管道 (pipeLine) 中。
- **transferBlock()**: 将指定数据块复制 (transfer) 到另一个 **Datanode** 上。数据块复制操作是在数据流管道中的数据节点出现故障，需要用新的数据节点替换异常的数据节

点时, DFSCClient 调用这个方法将数据流管道中正常数据节点上已经写入的数据块复制到新添加的数据节点上。

- **replaceBlock():** 将从源 Datanode 复制来的数据块写入本地 Datanode 上。写成功后通知 Namenode, 并且删除源 Datanode 上的数据块。这个操作主要用在数据块平衡操作 (balancing) 的场景下。
- **copyBlock():** 复制当前 Datanode 上的数据块。这个操作主要用在数据块平衡操作的场景下。
- **blockChecksum():** 获取指定数据块的校验值。
- **requestShortCircuitFds():** 获取一个短路 (short circuit) 读取数据块的文件描述符 (请参考文件短路读操作小节)。
- **releaseShortCircuitFds():** 释放一个短路读取数据块的文件描述符。
- **requestShortCircuitShm():** 获取保存短路读取数据块的共享内存。

4.5.2 Sender 和 Receiver

如图 4-38 所示, DataTransferProtocol 有两个子类——Sender 和 Receiver。其中 Sender 类封装了 DataTransferProtocol 的调用操作, 用于发起流式接口请求; Receiver 类封装了 DataTransferProtocol 的执行操作, 用于响应流式接口请求。

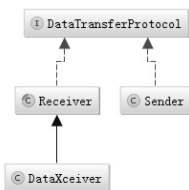


图 4-38 DataTransferProtocol 继承结构图

1. DataTransferProtocol 调用流程

如图 4-39 所示, 假设 DFSCClient 发起了一个 DataTransferProtocol.readBlock()操作, 那么 DFSCClient 会调用 Sender 类将这个请求序列化, 并传输给远程的 Receiver。远程的 Receiver 类接收到这个请求后, 会反序列化请求, 然后调用执行代码执行读取操作。

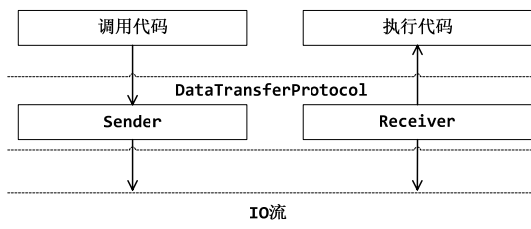


图 4-39 DataTransferProtocol 调用流程图

下面我们就学习 `Sender` 类以及 `Receiver` 类的实现。

2. `Sender` 类

`Sender` 类用于发起 `DataTransferProtocol` 请求。我们知道 `DataTransferProtocol` 接口定义的方法参数都比较多，`Sender` 类首先使用 `ProtoBuf` 将参数序列化，然后用一个枚举类 `Op` 描述调用的是什么方法，最后将序列化后的参数和 `Op` 一起发送给接收方。这里我们以 `readBlock()` 方法为例，学习 `Sender` 是如何发送流式接口请求的。`Sender` 类中其他方法的实现与 `readBlock()` 方法类似。

```
public void readBlock(final ExtendedBlock blk,
    final Token<BlockTokenIdentifier> blockToken,
    final String clientName,
    final long blockOffset,
    final long length,
    final boolean sendChecksum,
    final CachingStrategy cachingStrategy) throws IOException {
    // 将所有 DataTransferProtocol.readBlock() 方法中的参数用 ProtoBuf 序列化
    OpReadBlockProto proto = OpReadBlockProto.newBuilder()
        .setHeader(DataTransferProtoUtil.buildClientHeader(blk, clientName, blockToken))
        .setOffset(blockOffset)
        .setLen(length)
        .setSendChecksums(sendChecksum)
        .setCachingStrategy(getCachingStrategy(cachingStrategy))
        .build();

    // 调用 send() 方法发送 Op.READ_BLOCK 描述当前调用的是 readBlock() 方法
    // 同时发送序列化后的参数 proto
    send(out, Op.READ_BLOCK, proto);
}
```

可以看到，这里调用了 `send()` 方法将 `Op` 对象以及序列化后的参数发送到 IO 流中。`send()` 方法会先往 IO 流中写入一个 `short` 长度的 `DataTransferProtocol` 版本号，然后再写入操作码 `Op`，最后写入序列化后的参数。`send()` 方法的代码如下：

```
private static void send(final DataOutputStream out, final Op opcode,
    final Message proto) throws IOException {
    // 调用 op() 方法写入版本号，然后再写入操作码 Op
    op(out, opcode);
    // 写入序列化后的参数
    proto.writeDelimitedTo(out);
    out.flush();
}

// op() 方法用于向输出流中写入 DataTransferProtocol 版本号，然后再写入操作码 Op
private static void op(final DataOutput out, final Op op
    ) throws IOException {
    out.writeShort(DataTransferProtocol.DATA_TRANSFER_VERSION);
}
```

```

    op.write(out);
}

```

下面我们看一下 Op 的定义。Op 是一个枚举类型，使用一个 byte 类型的变量 code 标识操作码。一个操作码对应 DataTransferProtocol 接口中的一个方法，例如操作码 80 对应 DataTransferProtocol.writeBlock()方法。Op 的定义如下代码所示。

```

public enum Op {
    WRITE_BLOCK((byte)80), // 操作码 80, 对应 DataTransferProtocol.writeBlock() 方法
    READ_BLOCK((byte)81), // 操作码 81, 对应 DataTransferProtocol.readBlock() 方法
    READ_METADATA((byte)82), // 以下字段类似, 每个操作码都对应 DataTransferProtocol 的一个方法
    REPLACE_BLOCK((byte)83),
    COPY_BLOCK((byte)84),
    BLOCK_CHECKSUM((byte)85),
    TRANSFER_BLOCK((byte)86),
    REQUEST_SHORT_CIRCUIT_FDS((byte)87),
    RELEASE_SHORT_CIRCUIT_FDS((byte)88),
    REQUEST_SHORT_CIRCUIT_SHM((byte)89);

    // 操作码
    public final byte code;

    private Op(byte code) {
        this.code = code;
    }
    ...
}

```

这里总结一下，当我们通过调用 Sender 类发起一个 readBlock()操作时，Sender 类会将读取数据块的请求通过 IO 流发送给远程的 Datanode。Datanode 接收到这个请求后，会调用 Receiver 类的对应方法执行 readBlock()操作。读取数据块请求的格式如下所示，首先是一个 short 类型的 DataTransferProtocol 版本号，然后是 byte 类型的 Op 操作码，最后是通过 ProtoBuf 序列化的 readBlock()请求参数。DataTransferProtocol 中其他方法的格式与 readBlock()类似。

```

+-----+
| short -DataTransferProtocol 版本号| byte -Op 操作码 (OpCode) |方法的序列化参数|
+-----+

```

3. Receiver 类

Receiver 类封装了 DataTransferProtocol 的执行操作，用于执行远程节点发起的流式接口请求。Receiver 是一个抽象类，它提供了解析 Sender 请求操作码的 readOp()方法，以及处理 Sender 请求的 processOp()方法。这两个方法都是在 DataXceiver.run()中循环调用的，DataXceiver 的实现请参考下一节。

我们先学习 readOp()方法的代码实现。readOp()用于从 IO 流中获取 Op 操作码，代码如下：

```

protected final Op readOp() throws IOException {

```

Hadoop 2.X HDFS 源码剖析

```
// 先从数据流中读入 DataTransferProtocol 版本号，并与当前版本号进行比对
final short version = in.readShort();
if (version != DataTransferProtocol.DATA_TRANSFER_VERSION) {
    throw new IOException( "Version Mismatch...");
}
// 然后从数据流中读入 Op，并返回
return Op.read(in);
}
```

`processOp()`方法接收 `readOp()`解析出的 `Op` 操作码作为参数，然后针对不同的操作码调用指定的方法。这里以 `Op.READ_BLOCK` 操作为例，`processOp()`首先判断传入的 `Op` 是一个 `Op.READ_BLOCK` 事件，然后会调用 `opReadBlock()`方法处理。`processOp()`方法的代码如下：

```
protected final void processOp(Op op) throws IOException {
    switch(op) { // 根据不同的 Op 操作码调用指定的方法响应
    case READ_BLOCK:
        opReadBlock(); // 对于 Op.READ_BLOCK，调用 opReadBlock() 响应
        break;
    case WRITE_BLOCK:
        opWriteBlock(in); // 对于 Op.WRITE_BLOCK，调用 opWriteBlock () 响应
        break;
    case REPLACE_BLOCK:
        opReplaceBlock(in); // 对于 Op.REPLACE_BLOCK，调用 opReplaceBlock () 响应
        break;
    // ...省略
    default:
        throw new IOException("Unknown op " + op + " in data stream");
    }
}
```

`opReadBlock()`方法首先从 IO 流中解析出序列化的 `Receiver.readBlock()`方法的参数，然后对解析出的参数进行反序列化，最后调用 `Receiver.readBlock()`方法执行读取操作。这里的 `readBlock()`方法是在 `Receiver` 的子类 `DataXceiver` 中实现的。`opReplaceBlock()`、`OpWriteBlock()`等方法的实现与 `opReadBlock()`的实现类似，这里我们只学习 `opReadBlock()`的实现。

```
private void opReadBlock() throws IOException {
    // 从 IO 流中读取序列化的 readBlock() 参数
    OpReadBlockProto proto = OpReadBlockProto.parseFrom(vintPrefixed(in));
    TraceScope traceScope = continueTraceSpan(proto.getHeader(),
        proto.getClass().getSimpleName());
    try {
        // 反序列化参数，然后调用子类 DataXceiver 的 readBlock() 方法执行读取操作
        readBlock(PBHelper.convert(proto.getHeader().getBaseHeader().getBlock()),
            PBHelper.convert(proto.getHeader().getBaseHeader().getToken()),
            proto.getHeader().getClientName(),
            proto.getOffset(),
            proto.getLen(),
            proto.getSendChecksums(),
            (proto.hasCachingStrategy() ?
```



```

        getCachingStrategy(proto.getCachingStrategy()) :
        CachingStrategy.newDefaultStrategy());
    } finally {
        if (traceScope != null) traceScope.close();
    }
}

```

4.5.3 DataXceiverServer

我们知道在 Java 的 Socket 实现中，首先需要创建一个 `java.net.ServerSocket` 对象，绑定到某个指定的端口，然后通过 `ServerSocket.accept()` 方法监听是否有连接请求到达这个端口。当有 Socket 连接请求时，`ServerSocket.accept()` 方法会返回一个 `Socket` 对象，之后服务器就可以通过这个 `Socket` 对象与客户端通信了。如图 4-40 所示，Datanode 的流式接口就参考了 Socket 的实现，设计了 `DataXceiverServer` 以及 `DataXceiver` 两个对象，其中 `DataXceiverServer` 对象用于在 Datanode 上监听流式接口的请求，每当有 Client 通过 `Sender` 类发起流式接口请求时，`DataXceiverServer` 就会监听并接收这个请求，然后创建一个 `DataXceiver` 对象用于响应这个请求并执行对应的操作。本节我们就学习 `DataXceiverServer` 的实现。

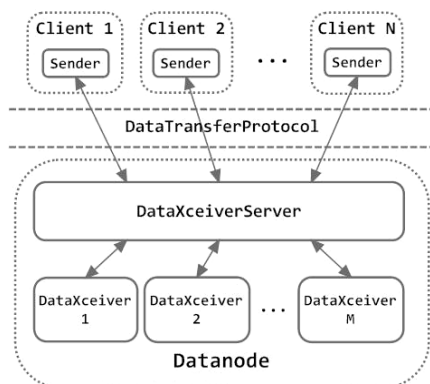


图 4-40 DataXceiverServer 逻辑结构图

1. DataXceiverServer 的初始化

在 `DataNode` 的初始化代码中，会创建一个 `DataXceiverServer` 对象监听所有流式接口请求，`Datanode` 会调用 `Datanode.initDataXceiver()` 方法来完成 `DataXceiverServer` 对象的构造。`initDataXceiver()` 方法会首先创建 `TcpPeerServer` 对象（对 `ServerSocket` 的封装），它能够通过 `accept()` 方法返回 `Peer` 对象（封装了 `Socket` 对象，提供通信的输入/输出流）。`TcpPeerServer` 的监听地址是通过 `dfs.datanode.address` 配置项配置的，`TcpPeerServer` 会将 `Socket` 设置为 `No_Delay` 的方式，然后设置 `receiveBuffer` 为 128KB（默认为 8KB），同时将 `DataXceiverServer` 放入线程组 `dataXceiverServer` 中，最后将这个线程组设置为守护线程。

注：（1）`ThreadGroup` 表示一个线程的集合，可以通过线程组的函数对一组线程进行操作，例如 `setDaemon()`。

（2）守护线程：如果虚拟机中只有守护线程在运行，则虚拟机退出；如果存在一个用户线程，则守护线程仍然执行。

`initDataXceiver()` 方法的第二部分代码则是与短路操作相关的，如果用户通过设置 `dfs.client.read.shortcircuit` 或者 `dfs.client.domain.socket.data.traffic` 配置项开启了短路操作，那

么 `DataNode` 会创建一个 `localDataXceiverServer` 响应本地的短路读取请求。这里要特别注意, `localDataXceiverServer` 底层的 `PeerServer` 为 `DomainPeerServer`——也就是通信时并不使用基于 TCP 连接的传统的 `TcpPeerServer` (底层封装 `Socket`), 而是使用 `DomainPeerServer` (底层基于 `DomainSocket`)。

注: 在 Linux 中, 有种技术叫作 UNIX Domain Socket。UNIX Domain Socket 是一种进程间的通信方式, 它使得同一台机器上的两个进程能以 `Socket` 的方式通信。它带来的另一大好处是, 利用它, 两个进程除了可以传递普通数据外, 还可以在进程间传递文件描述符。通过这种技术, 就可以实现 `DataNode` 与 `DFSClient` 的本地读取功能。

`initDataXceiver()`方法的代码如下:

```
private void initDataXceiver(Configuration conf) throws IOException {

    TcpPeerServer tcpPeerServer;
    // 创建 TcpPeerServer
    if (secureResources != null) {
        tcpPeerServer = new TcpPeerServer(secureResources);
    } else {
        tcpPeerServer = new TcpPeerServer(dnConf.socketWriteTimeout,
            DataNode.getStreamingAddr(conf));
    }
    // 设置 TCP 接收缓冲区
    tcpPeerServer.setReceiveBufferSize(HdfsConstants.DEFAULT_DATA_SOCKET_SIZE);
    streamingAddr = tcpPeerServer.getStreamingAddr();
    this.threadGroup = new ThreadGroup("dataXceiverServer");
    // 构造 DataXceiverServer 对象
    xserver = new DataXceiverServer(tcpPeerServer, conf, this);
    // 将 DataXceiverServer 线程组设置为守护线程
    this.dataXceiverServer = new Daemon(threadGroup, xserver);
    this.threadGroup.setDaemon(true); // auto destroy when empty

    // 短路读取情况
    if (conf.getBoolean(DFSConfigKeys.DFS_CLIENT_READ_SHORTCIRCUIT_KEY,
        DFSConfigKeys.DFS_CLIENT_READ_SHORTCIRCUIT_DEFAULT) ||
        conf.getBoolean(DFSConfigKeys.DFS_CLIENT_DOMAIN_SOCKET_DATA_TRAFFIC,
            DFSConfigKeys.DFS_CLIENT_DOMAIN_SOCKET_DATA_TRAFFIC_DEFAULT)) {
        // 构造 DomainPeerServer (底层基于 DomainSocket, 用于本地进程间通信)
        DomainPeerServer domainPeerServer =
            getDomainPeerServer(conf, streamingAddr.getPort());
        if (domainPeerServer != null) {
            // 构造 localDataXceiverServer
            this.localDataXceiverServer = new Daemon(threadGroup,
                new DataXceiverServer(domainPeerServer, conf, this));
        }
    }
}
```

```
// 创建 ShortCircuitRegistry 对象 (请参考第 5 章的文件短路读操作小节)
this.shortCircuitRegistry = new ShortCircuitRegistry(conf);
}
```

2. run()方法

DataXceiverServer 的功能都是在 run()方法中实现的。DataXceiverServer.run()方法的逻辑非常简单,它会循环调用 peerServer 的 accept()方法监听,如果有新的连接请求则创建 Peer 对象,并构造一个 DataXceiver 线程服务这个流式请求——也就是 DataXceiverServer 只负责连接的建立以及构造并启动 DataXceiver,流式接口请求则是由 DataXceiver 响应的,真正的操作都是由 DataXceiver 来进行的。

注:无论是在 Hadoop RPC 接口的实现,还是流式接口的实现中,都采用了连接建立和响应分离这种设计方式,值得我们积累。

run()方法会捕获 5 种异常。这里我们重点说一下 AsynchronousCloseException,当通过 DataXceiverServer.kill()关闭服务器时,会触发这种异常。这是因为 DataXceiverServer 常规的关闭方法是 will dataNode.shouldRun 字段设置为 false,这样 run()方法通过循环时的检查就可以正常退出。但是当 dataNode.shouldRun 字段已经被设置为 false,而 DataXceiverServer 却在 accept()中阻塞时,就需要调用 kill()方法强行将 ServerSocket 关闭。这种情况就会抛出 AsynchronousCloseException 异常,这时就需要强行结束 DataXceiverServer 的服务了。

当 DataXceiverServer 退出主循环后,就会进行 DataXceiverServer 的关闭清理操作,这种设计模式在 Hadoop RPC 框架的实现中也使用了。在线程的 run()方法中携带了清理程序,通过标志位来进行流程控制,退出循环之后在 run()方法中进行程序的清理,请读者注意积累。run()方法的代码如下:

```
public void run() {
    Peer peer = null;
    // 循环执行逻辑
    while (datanode.shouldRun && !datanode.shutdownForUpgrade) {
        try {
            peer = peerServer.accept(); // 接受连接请求
            int curXceiverCount = datanode.getXceiverCount();
            if (curXceiverCount > maxXceiverCount) {
                throw new IOException("Xceiver count " + curXceiverCount + "...");
            }
            // 创建一个 DataXceiver 响应请求
            new Daemon(datanode.threadGroup, DataXceiver.create(peer, datanode, this)).start();
        } catch (SocketTimeoutException ignored) {
            // Socket 超时异常直接忽略
        } catch (AsynchronousCloseException ace) {
            if (datanode.shouldRun && !datanode.shutdownForUpgrade) {
                LOG.warn(datanode.getDisplayName() + ":DataXceiverServer: ", ace);
            }
        }
    }
}
```

```
    } catch (IOException ie) {
        IOUtils.cleanup(null, peer);
    } catch (OutOfMemoryError ie) {
        IOUtils.cleanup(null, peer);
        try {
            Thread.sleep(30 * 1000);
        } catch (InterruptedException e) {
        }
    } catch (Throwable te) {
        // 其他异常则直接关闭 Datanode
        datanode.shouldRun = false;
    }
}

// 清理操作: 退出主循环, 执行关闭操作, 将 peerServer 关闭
try {
    peerServer.close();
    closed = true;
} catch (IOException ie) {
}
// ...
// ...将当前 Server 下的所有连接也全部关闭, 并且清理存储连接的数据结构
closeAllPeers();
}
```

4.5.4 DataXceiver

通过上一节的介绍我们知道, `DataXceiverServer` 主要用于监听并接收流式请求, 然后创建并启动 `DataXceiver` 对象。`DataXceiver` 是 `Receiver` 的子类, `DataTransferProtocol` 真正的响应操作都是在 `DataXceiver` 类中实现的。本节我们就重点学习 `DataXceiver` 类的实现。

1. run()方法

`DataXceiver` 的执行逻辑主要是在 `run()`方法中完成的。`DataXceiver.run()`方法首先根据传入的 `Peer` 对象获取此层网络的输入/输出流, 并对输入/输出流进行装饰, 然后调用父类 `Receiver` 的 `initialize()`方法执行初始化操作。

```
dataXceiverServer.addPeer(peer, Thread.currentThread(), this);
peer.setWriteTimeout(datanode.getDnConf().socketWriteTimeout);
InputStream input = socketIn; // 获取底层的输入流
try {
    IOStreamPair saslStreams = datanode.saslServer.receive(peer, socketOut,
        socketIn, datanode.getXferAddress().getPort(),
        datanode.getDatanodeId());
    input = new BufferedInputStream(saslStreams.in,
        HdfsConstants.SMALL_BUFFER_SIZE); // 对输入流进行装饰
    socketOut = saslStreams.out; // 获取底层的输出流
}
```

```

    } catch (InvalidMagicNumberException imne) {
        return;
    }
    // 调用父类的 initialize() 方法完成 Receiver 的初始化操作
    super.initialize(new DataInputStream(input));

```

接下来 `run()` 方法会循环读取 IO 流中的流式接口请求，并调用对应的方法响应请求。这里注意，对于操作符 `Op` 的解析是通过调用父类 `Receiver.readOp()` 方法完成的，对于操作符的处理是通过调用父类 `Receiver.processOp()` 方法完成的，`processOp()` 方法会根据 `Op` 调用 `DataXceiver` 对应的处理方法（我们在 `Receiver` 类小节中已经给出了分析）。

```

do {
    updateCurrentThreadName("Waiting for operation #" + (opsProcessed + 1));
    try {
        if (opsProcessed != 0) {
            assert dnConf.socketKeepaliveTimeout > 0;
            peer.setReadTimeout(dnConf.socketKeepaliveTimeout);
        } else {
            peer.setReadTimeout(dnConf.socketTimeout);
        }
        op = readOp(); // 调用 Receiver.readOp() 从输入流中解析操作符
    } catch (InterruptedException ignored) {
        // 等待 Client RPC 超时
        break;
    } catch (IOException err) {
        if (opsProcessed > 0 &&
            (err instanceof EOFException || err instanceof ClosedChannelException)) {
        } else {
            throw err;
        }
        break;
    }
    // ...
    opStartTime = now();
    processOp(op); // 调用 processOp() 处理这个流式请求
    ++opsProcessed;
} while ((peer != null) &&
    (!peer.isClosed() && dnConf.socketKeepaliveTimeout > 0));

```

2. DataTransferProtocol 的实现

在 `Receiver` 类小节中我们介绍了，`Receiver.processOp()` 方法用于处理流式接口的请求，它首先从数据流中读取序列化后的参数，对参数反序列化，然后根据操作码调用 `DataTransferProtocol` 中定义的方法，这些方法都是在 `DataXceiver` 中具体实现的。

由于 `DataXceiver` 实现 `DataTransferProtocol` 定义的方法都比较复杂，所以我们将 `DataXceiver` 实现的方法分为 4 类：读数据、写数据、短路读取以及其他方法。下面我们将介绍这 4 类方法的实现。

4.5.5 读数据

流式接口中最重要的一个部分就是客户端从数据节点上读取数据块，`DataTransferProtocol.readBlock()`给出了读取操作的接口定义，操作码是 81。`DataXceiver.readBlock()`则实现了 `DataTransferProtocol.readBlock()`方法。

如图 4-41 所示，客户端通过调用 `Sender.readBlock()`方法从指定数据节点上读取数据块，请求通过 IO 流到达数据节点后，数据节点的 `DataXceiverServer` 会创建一个 `DataXceiver` 对象响应流式接口请求。`DataXceiver.processOp()`方法解析操作码为 81（读请求），则调用 `DataXceiver.readBlock()`响应这个读请求。

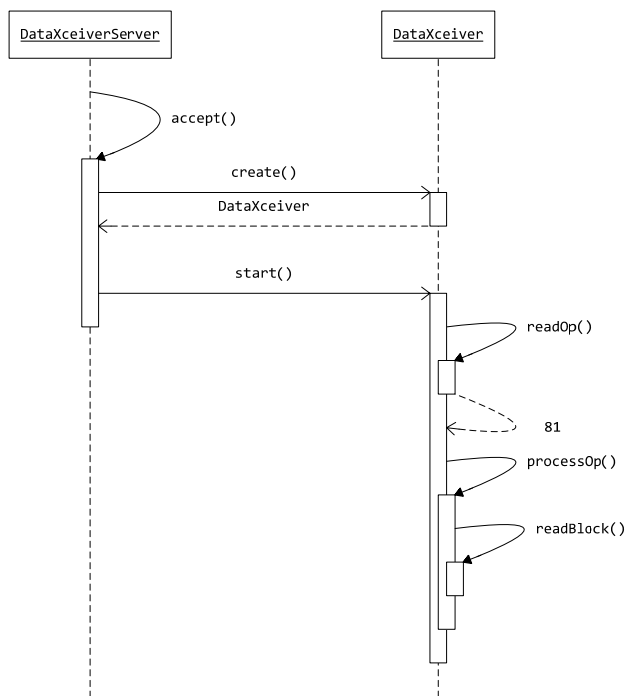


图 4-41 `DataTransferProtocol.readBlock()`调用流程图

`DataXceiver.readBlock()`首先向客户端回复一个 `BlockOpResponseProto` 响应，指明请求已经成功接收，并通过 `BlockOpResponseProto` 响应给出 `Datanode` 当前使用的校验方式。接下来 `DataXceiver.readBlock()`方法会将数据节点上的数据块（block）切分成若干个数据包（packet），然后依次将数据包发送给客户端。客户端会在收到每个数据包时进行校验，如果校验和错误，客户端会切断与当前数据节点的连接，选择新的数据节点读取数据；如果数据块内的所有数据包都校验成功，客户端会给数据节点发送一个 `Status.CHECKSUM_OK` 响应，表明读取成功。读取数据流程如图 4-42 所示。

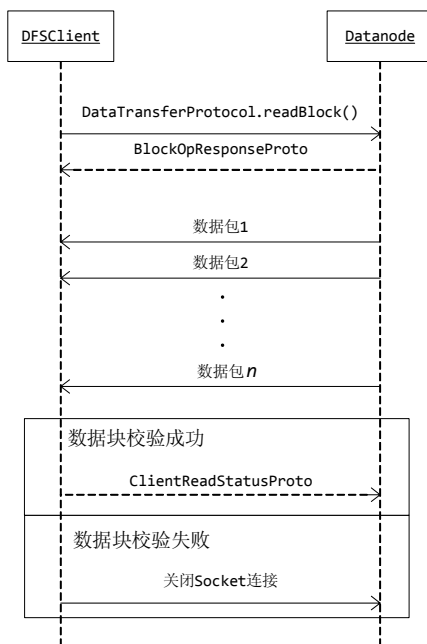


图 4-42 读取数据流程图

1. DataXceiver.readBlock()

上面描述的数据块读取操作的逻辑都是在 `DataXceiver.readBlock()` 方法中实现的，本节我们就学习 `readBlock()` 方法的实现。首先看一下 `DataTransferProtocol.readBlock()` 方法的定义。

```

public void readBlock(final ExtendedBlock block,
    final Token<BlockTokenIdentifier> blockToken,
    final String clientName,
    final long blockOffset,
    final long length,
    final boolean sendChecksum,
    final CachingStrategy cachingStrategy) throws IOException {

```

`readBlock()` 需要传入的参数主要有如下几个。

- `ExtendedBlock blk`: 要读取的数据块。
- `Token<BlockTokenIdentifier> blockToken`: 数据块的访问令牌。
- `String clientName`: 客户端的名称。
- `long blockOffset`: 要读取数据在数据块中的位置。
- `long length`: 读取数据的长度。
- `sendChecksum`: `Datanode` 是否发送校验数据，如果为 `false`，则 `Datanode` 不发送校验数据。这里要特别注意，数据块的读取校验工作是在客户端完成的，客户端会将校验结果返回给 `Datanode`。

- **CachingStrategy cachingStrategy**: 缓存策略, 这里主要包括两个重要的字段——**readahead**, 预读取操作, **Datanode** 会在读取数据块文件时预读取部分数据至操作系统缓存中, 以提高读取文件效率 (请参考 **BlockSender** 实现的预读取&丢弃小节); **dropBehind**, 如果缓存中存放的文件比较多, 那么在读取完数据之后, 就马上从缓存中将数据删除 (请参考 **BlockSender** 实现的预读取&丢弃小节)。
- **readBlock()**方法的执行流程如图 4-43 所示, 分为如下几个步骤。
- 创建 **BlockSender** 对象: 首先调用 **getOutputStream()**方法获取 **Datanode** 连接到客户端的 IO 流, 然后构造 **BlockSender** 对象。
- 成功创建 **BlockSender** 对象后, 调用 **writeSuccessWithChecksumInfo()** 方法发送 **BlockOpResponseProto** 响应给客户端, 通知客户端读请求已经成功接收, 并且告知客户端当前数据节点的校验信息。
- 调用 **BlockSender.sendBlock()**方法将数据块发送给客户端。**BlockSender** 类我们在下面的 **BlockSender** 实现小节中学习。
- 当 **BlockSender** 完成发送数据块的所有内容后, 客户端会响应一个状态码, **Datanode** 需要解析这个状态码。

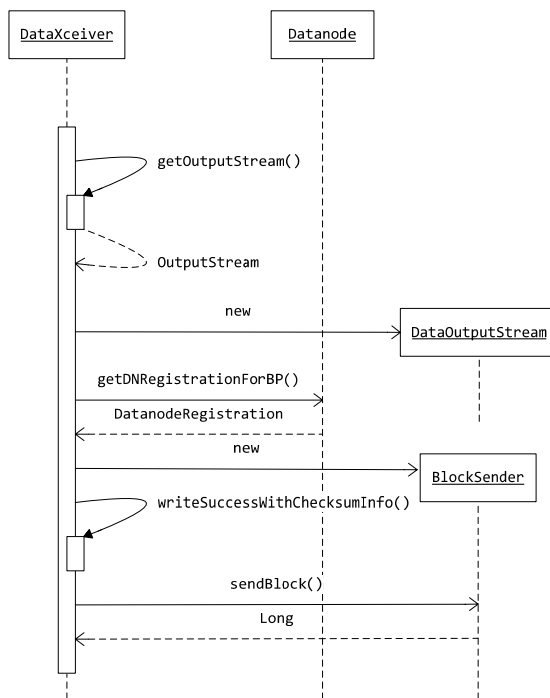


图 4-43 readBlock()执行流程图

readBlock()的异常处理逻辑也比较简单, 当客户端关闭了当前 **Socket** (可能是出现了校验错误), 或者无法从 IO 流中成功获取客户端发回的响应时, 则直接关闭 **Datanode** 到客户端

的底层输出流。在 readBlock()方法的最后，会关闭 BlockSender 类以执行清理操作。readBlock()方法的逻辑比较简单，这里就不贴出代码了，请读者结合图 4-43 自行参考源码学习。

2. 数据块的传输格式

BlockSender 类主要负责从数据节点的磁盘读取数据块，然后发送数据块到接收方。需要注意的是，BlockSender 发送的数据是以一定结构组织的。在介绍 BlockSender 的实现之前，我们首先了解数据块的传输格式。BlockSender 发送数据的格式如图 4-44 所示。

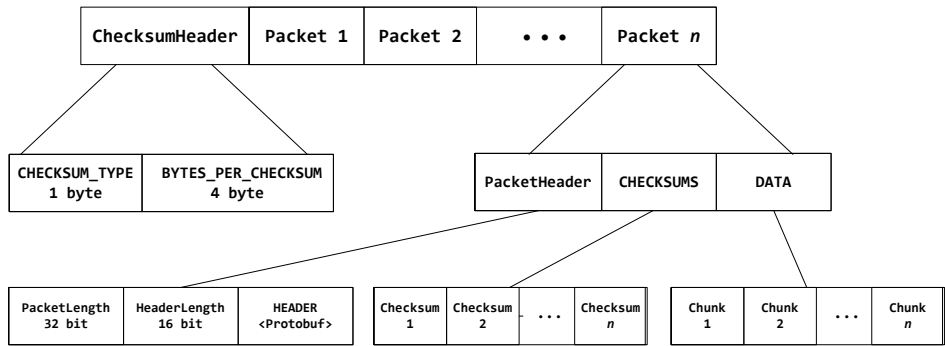


图 4-44 发送数据格式

这里要特别注意，PacketLength 大小为：4 + CHECKSUMS（校验数据的大小）+ DATA（真实数据的大小）。这里为什么要加上 4 呢？文档中给出的说明是历史原因，在处理时需要处理一下。HeaderLength 大小为 HEADER 序列化完成后的长度。

下面我们来详解这个数据格式。BlockSender 发送数据的格式包括两个部分：校验信息头（ChecksumHeader）和数据包序列（packets）。

```
+-----+
| 校验信息头 (ChecksumHeader) | 数据包序列 (packets) |
+-----+
```

我们分别看一下这两个部分。

(1) 校验信息头（ChecksumHeader）

ChecksumHeader 是一个校验信息头，用于描述当前 Datanode 使用的校验方式等信息。如下所示，一个校验信息头的结构也包含两个部分。

```
+-----+
| 1 byte 校验类型 (CHECKSUM_TYPE) | 4 byte 校验块大小 (BYTES_PER_CHECKSUM) |
+-----+
```

- 数据校验类型：数据校验类型定义在 org.apache.hadoop.util.DataChecksum 中，目前包括三种方式——空校验（不进行校验）、CRC32 以及 CRC32C。这里使用 1 byte 描述数据校验类型，空校验、CRC32、CRC32C 分别对应于值 0、1、2。
- 校验块大小：校验信息头中的第二个部分是校验块的大小，也就是多少字节的数据

产生一个校验值。这里以 CRC32 为例，一般情况下是 512 字节的数据产生一个 4 字节的校验和，我们把这 512 字节的数据称为一个校验块（**chunk**）。这个校验块的概念非常重要，它是 HDFS 中读取和写入数据块操作的最小单元，在后面的章节中会多次提到这个概念。

（2）数据包序列

BlockSender 会将数据块切分成若干数据包（**packet**）对外发送，当数据发送完成后，会以一个空的数据包作为结束。如下所示，每个数据包都包括一个变长的包头、校验数据以及若干字节的实际数据。

```
+-----+
| 变长的数据包头 (packetHeader) |
+-----+
| 校验数据                      |
+-----+
| 实际数据 .....              |
+-----+
```

我们依次看一下这三个部分数据的格式。

- **数据包头**——数据包头用于描述当前数据包的信息，是通过 **ProtoBuf** 序列化的，包括 4 字节的全包长度，以及 2 字节的包头长度，之后紧跟如下数据包信息。
 - 当前数据包在整个数据块中的位置。
 - 数据包在管道中的序列号。
 - 当前数据包是不是数据块中的最后一个数据包。
 - 当前数据包中数据部分的长度。
 - 是否需要 **DN** 同步。
- **校验数据**——校验数据是对实际数据做校验操作产生的，它将实际数据以校验块为单位，每个校验块产生一个校验和，校验数据中包含了所有校验块的校验和。校验数据的大小为： $(\text{实际数据长度} + \text{校验块大小}) / \text{校验块大小} \times \text{校验和长度}$ 。
- **实际数据**——数据包中的实际数据就是数据块文件中保存的数据，实际数据的传输是以校验块为单位的，一个校验块对应产生一个校验和的实际数据。在数据包中会将校验块与校验数据分开发送，首先将所有校验块的校验数据发送出去，然后再发送所有的校验块。

3. BlockSender 实现

数据块的发送主要是由 **BlockSender** 类执行的，我们在这一小节中学习 **BlockSender** 的实现。**BlockSender** 中数据块的发送过程包括：发送准备、发送数据块以及清理工作。**BlockSender** 的代码比较长，这里会将代码分块介绍，请读者注意。

（1）发送准备——构造方法

BlockSender 中发送数据的准备工作主要是在 **BlockSender** 的构造方法中执行的，

BlockSender 的构造方法执行了以下操作。

- **readahead & dropBehind 的处理：**如果用户通过 cachingStrategy 设置了这两个字段，则按照这两个字段初始化读取操作。如果 cachingStrategy 为 Null，则按照配置文件设置 dropCacheBehindLargeReads 为 dfs.datanode.drop.cache.behind.reads，设置 readaheadLength 为 dfs.datanode.readahead.bytes，默认为 4MB。
- **赋值与校验：**检查当前 Datanode 上被读取数据块的时间戳、数据块文件的长度等状态是否正常。
- **是否开启 transferTo 模式：**默认为 true，transferTo 机制请参考零拷贝数据传输小节内容。
- **获取 checksum 信息：**从 Meta 文件中获取当前数据块的校验算法、校验和长度，以及多少字节产生一个校验值，也就是校验块的大小。

```
chunkSize = size;    // 校验块大小
checksum = csum;     // 校验算法
checksumSize = checksum.getChecksumSize(); // 校验和长度
```

- **计算 offset 以及 endOffset：**offset 变量用于标识要读取的数据在数据块的起始位置，endOffset 则用于标识结束的位置。由于读取位置往往不会落在某个校验块的起始位置，所以在准备工作中需要确保 offset 在校验块的起始位置，endOffset 在校验块的结束位置。这样读取时就可以以校验块为单位读取，方便校验和的操作。

```
// 将 offset 位置设置在校验块的边界上，也就是校验块的起始位置
offset = startOffset - (startOffset % chunkSize);
if (length >= 0) {
    // 计算 endOffset 的位置，确保 endOffset 在校验块的结束位置
    long tmpLen = startOffset + length;
    if (tmpLen % chunkSize != 0) {
        tmpLen += (chunkSize - tmpLen % chunkSize);
    }
    if (tmpLen < end) {
        // 结束位置还在数据块内，则可以使用磁盘上的校验值
        end = tmpLen;
    } else if (chunkChecksum != null) {
        // 目前有写线程正在更改这个校验块，则使用内存中的校验值
        this.lastChunkChecksum = chunkChecksum;
    }
}
endOffset = end;
```

- **将数据块文件与校验和文件的 offset 都移动到指定位置。**

```
//将校验文件的坐标移动到 offset 对应的位置
if (offset > 0 && checksumIn != null) {
    long checksumSkip = (offset / chunkSize) * checksumSize;
    if (checksumSkip > 0) {
        IOUtils.skipFully(checksumIn, checksumSkip);
    }
}
```

```
// packet 序列号设置为 0
seqno = 0;
// 将数据块文件的坐标移动到 offset 位置
blockIn = datanode.data.getBlockInputStream(block, offset);
if (blockIn instanceof FileInputStream) {
    blockInFd = ((FileInputStream)blockIn).getFD();
} else {
    blockInFd = null;
}
```

(2) 预读取&丢弃——manageOsCache()

BlockSender 在读取数据块之前，会先调用 manageOsCache()方法执行预读取(read-ahead)操作以提高读取效率。预读取操作就是将数据块文件提前读取到操作系统的缓存中，这样当 BlockSender 到文件系统中读取数据块文件时，可以直接从操作系统的缓存中读取数据，比直接从磁盘上读取快很多。但是操作系统的缓存空间是有限的，所以需要调用 manageOsCache()方法将不再使用的数据从缓存中丢弃(drop-behind)，为新的数据挪出空间。BlockSender 在读取数据时，使用了预读取以及丢弃这两个特性，下面介绍这两种操作的实现。

manageOsCache()方法在 HDFS 管理员设置了预读取的长度（默认是 4MB）并且设置了所有的操作都使用预读取时，或者当前读取是一个长读取（超过 256KB 的读取）时，会调用 ReadaheadPool.readaheadStream()方法触发一个预读取操作，这个预读取操作会从磁盘文件上预读取部分数据块文件的数据到操作系统的缓存中。同时 managerOsCache()还会处理丢弃操作，如果 dropCacheBehindAllReads（所有读操作后都丢弃）为 true 或者当前读取是一个大读取时，则触发丢弃操作。manageOsCache()方法会判断如果下一次读取数据的坐标 offset 大于下一次丢弃操作的开始坐标，则将 lastCacheDropOffset（上一次丢弃操作的结束位置）和 offset 之间的数据全部从缓存中丢弃，因为这些数据 Datanode 已经读取了，不需要放在缓存中了。manageOsCache()方法的代码如下：

```
private void manageOsCache() throws IOException {
    if (blockInFd == null) return;

    // 按条件触发预读取操作
    if ((readaheadLength > 0) && (datanode.readaheadPool != null) &&
        (alwaysReadahead || isLongRead())) {
        // 满足预读取条件，则调用 ReadaheadPool.readaheadStream()方法触发预读取
        curReadahead = datanode.readaheadPool.readaheadStream(
            clientTraceFmt, blockInFd, offset, readaheadLength, Long.MAX_VALUE,
            curReadahead);
    }

    // 丢弃刚才从缓存中读取的数据，因为不再需要使用这些数据了
    if (dropCacheBehindAllReads ||
        (dropCacheBehindLargeReads && isLongRead())) {
        // 丢弃数据的位置
        long nextCacheDropOffset = lastCacheDropOffset + CACHE_DROP_INTERVAL_BYTES;
    }
}
```

```

if (offset >= nextCacheDropOffset) {
    // 如果下一次读取数据的位置大于丢弃数据的位置，则将读取数据位置前的数据全部丢弃
    long dropLength = offset - lastCacheDropOffset;
    NativeIO.POSIX.getCacheManipulator().posixFadviseIfPossible(
        block.getBlockName(), blockInFd, lastCacheDropOffset,
        dropLength, NativeIO.POSIX.POSIX_FADV_DONTNEED);
    lastCacheDropOffset = offset;
}
}
}

```

`ReadaheadPool.readaheadStream()`方法执行了一个预读取操作，只有在上一次预读取的数据已经使用了一半时，才会触发一次新的预读取。新的预读取操作是通过在 `Datanode.readaheadPool` 线程池中创建一个 `ReadaheadRequestImpl` 任务来执行的。`ReadaheadRequestImpl.run()`方法的代码如下，它通过调用 `fadvise()`系统调用，完成 OS 层面的预读取，将数据放入操作系统的缓存中。

```

public void run() {
    if (canceled) return;
    // 调用 fadvise() 系统调用完成预读取
    try {
        NativeIO.POSIX.getCacheManipulator().posixFadviseIfPossible(identifier,
            fd, off, len, NativeIO.POSIX.POSIX_FADV_WILLNEED);
    } catch (IOException ioe) {
        if (canceled) {
            return;
        }
        LOG.warn("Failed readahead on " + identifier,
            ioe);
    }
}
}

```

（3）发送数据块——`sendBlock()`

介绍完了预读取功能后，我们来学习 `BlockSender.sendBlock()`方法，这个方法用于读取数据以及校验和，并将它们发送到接收方。整个发送的流程可以分为如下几步。

- 在刚开始读取文件时，触发一次预读取，预读取部分数据到操作系统的缓冲区中。
- 构造 `pktBuf` 缓冲区，也就是能容纳一个数据包的缓冲区。这里首先要确定的就是 `pktBuf` 缓冲区的大小，最好是一个数据包的大小。对于两种不同的发送数据包的模式 `transferTo` 和 `ioStream`，缓冲区的大小是不同的。在 `transferTo` 模式中（请参考零拷贝数据传输小节），数据块文件是通过零拷贝方式直接传输给客户端的，并不需要将数据块文件写入缓冲区中，所以 `pktBuf` 缓冲区只需要缓冲校验数据即可；而 `ioStream` 模式则需要将实际数据以及校验数据都缓冲下来，所以 `pktBuf` 大小是完全不同的，读者需要注意。
- 接下来就是循环调用 `sendPacket()`方法发送数据包序列，直到 `offset >= endOffset`，也就是整个数据块都发送完成了。这里首先调用 `manageOsCache()`进行预读取，然后循

环调用 `sendPacket()` 依次将所有数据包发送到客户端。最后更新 `offset`——也就是数据游标，更新 `seqno`——记录已经发送了几个数据包。

- 发送一个空的数据包用以标识数据块的结束。
- 完成数据块发送操作之后，调用 `close()` 方法关闭数据块以及校验文件，并从操作系统的缓存中删除已读取的数据。

`sendBlock()` 方法的代码如下：

```
long sendBlock(DataOutputStream out, OutputStream baseStream,
               DataTransferThrottler throttler) throws IOException {
    // 1. 将数据预读取至操作系统的缓存中
    manageOsCache();
    final long startTime = ClientTraceLog.isDebugEnabled() ? System.nanoTime() : 0;

    // 2. 构造存放数据包 (packet) 的缓冲区
    try {
        int maxChunksPerPacket;
        int pktBufSize = PacketHeader.PKT_MAX_HEADER_LEN;
        boolean transferTo = transferToAllowed && !verifyChecksum
            && baseStream instanceof SocketOutputStream
            && blockIn instanceof FileInputStream;
        if (transferTo) {
            FileChannel fileChannel = ((FileInputStream)blockIn).getChannel();
            blockInPosition = fileChannel.position();
            streamForSendChunks = baseStream;
            // 这里的 TRANSFERTO_BUFFER_SIZE 大小默认是 64KB
            // maxChunksPerPacket 变量表明一个数据包中最多包含多少个校验块
            maxChunksPerPacket = numberOfChunks(TRANSFERTO_BUFFER_SIZE);
            // 缓冲区中只存放校验数据
            pktBufSize += checksumSize * maxChunksPerPacket;
        } else {
            // 这里的 IO_FILE_BUFFER_SIZE 大小默认是 4KB
            maxChunksPerPacket = Math.max(1,
                numberOfChunks(HdfsConstants.IO_FILE_BUFFER_SIZE));
            // 缓冲区存放校验数据以及实际数据
            pktBufSize += (chunkSize + checksumSize) * maxChunksPerPacket;
        }
        // 构造缓冲区 pktBuf
        ByteBuffer pktBuf = ByteBuffer.allocate(pktBufSize);

        // 循环调用 sendPacket() 发送 packet
        while (endOffset > offset && !Thread.currentThread().isInterrupted()) {
            manageOsCache();
            long len = sendPacket(pktBuf, maxChunksPerPacket, streamForSendChunks,
                transferTo, throttler);
            offset += len;
            totalRead += len + (numberOfChunks(len) * checksumSize);
            seqno++;
        }
    }
}
```

```

    }
    // 如果当前线程被中断, 则不再发送完整的数据块
    if (!Thread.currentThread().isInterrupted()) {
        try {
            // 发送一个空的数据包用以标识数据块的结束
            sendPacket(pktBuf, maxChunksPerPacket, streamForSendChunks, transferTo,
                throttler);
            out.flush();
        } catch (IOException e) { // socket error
            throw ioeToSocketException(e);
        }
        sentEntireByteRange = true;
    }
} finally {
    // 调用 close() 文件关闭数据块文件、校验文件以及回收操作系统缓冲区
    close();
}
return totalRead;
}

```

可以看到, 发送数据包主要是通过 `sendPacket()` 方法, 代码如下所示。`sendPacket()` 也可以分为三个部分。

- 首先计算数据包头域在 `pkt` 缓存中的位置 `headerOff`, 再计算 `checksum` 在 `pkt` 中的位置 `checksumOff`, 以及实际数据在 `pkt` 中的位置 `dataOff`。然后将数据包头域、校验数据以及实际数据写入 `pkt` 缓存中。如果 `verifyChecksum` 属性被设置为 `true`, 则调用 `verifyChecksum()` 方法确认校验和数据正确。
- 接下来就是发送数据块了, 将 `pkt` 缓存中的数据写入 IO 流中。这里要注意, 如果是 `transferTo` 方式, `pkt` 中只有数据包头域以及校验数据, 实际数据则直接通过 `transferTo` 方式从文件通道 (`FileChannel`) 直接写入 IO 流中。
- 使用节流器控制写入的速度。

```

private int sendPacket(ByteBuffer pkt, int maxChunks, OutputStream out,
    boolean transferTo, DataTransferThrottler throttler) throws IOException {
    int dataLen = (int) Math.min(endOffset - offset,
        (chunkSize * (long) maxChunks));
    int numChunks = numberOfChunks(dataLen); // 数据包中包含多少个校验块
    int checksumDataLen = numChunks * checksumSize; // 校验数据长度
    int packetLen = dataLen + checksumDataLen + 4; // 数据包长度
    boolean lastDataPacket = offset + dataLen == endOffset && dataLen > 0;
    int headerLen = writePacketHeader(pkt, dataLen, packetLen); // 将数据包头域写入缓存中

    int headerOff = pkt.position() - headerLen; // 数据包头域在缓存中的位置
    int checksumOff = pkt.position(); // 校验数据在缓存中的位置
    byte[] buf = pkt.array();

    if (checksumSize > 0 && checksumIn != null) {
        readChecksum(buf, checksumOff, checksumDataLen); // 将校验数据写入缓存中
    }
}

```

Hadoop 2.X HDFS 源码剖析

```
// ...
}

int dataOff = checksumOff + checksumDataLen;
if (!transferTo) { // 在普通模式下, 将实际数据写入缓存中
    IOUtils.readFully(blockIn, buf, dataOff, dataLen);
    if (verifyChecksum) { // 确认校验和正确
        verifyChecksum(buf, dataOff, dataLen, numChunks, checksumOff);
    }
}

try {
    if (transferTo) { // transferTo 模式
        SocketOutputStream sockOut = (SocketOutputStream)out;
        // 将头域和校验和写入输出流中
        sockOut.write(buf, headerOff, dataOff - headerOff);

        // 使用 transfer 方式, 将数据从数据块文件直接零拷贝到 IO 流中
        FileChannel fileCh = ((FileInputStream)blockIn).getChannel();
        LongWritable waitTime = new LongWritable();
        LongWritable transferTime = new LongWritable();
        sockOut.transferToFully(fileCh, blockInPosition, dataLen,
            waitTime, transferTime);
        datanode.metrics.addSendDataPacketBlockedOnNetworkNanos(waitTime.get());
        datanode.metrics.addSendDataPacketTransferNanos(transferTime.get());
        blockInPosition += dataLen;
    } else {
        // 在正常模式下
        // 将缓存中的所有数据 (包括头域、校验和以及实际数据) 写入输出流中
        out.write(buf, headerOff, dataOff + dataLen - headerOff);
    }
} catch (IOException e) {
    if (e instanceof SocketTimeoutException) {
    } else {
        String ioem = e.getMessage();
        if (!ioem.startsWith("Broken pipe") && !ioem.startsWith("Connection reset")) {
            LOG.error("BlockSender.sendChunks() exception: ", e);
        }
    }
    throw ioeToSocketException(e);
}

if (throttler != null) { // 调整节流器
    throttler.throttle(packetLen);
}
return dataLen;
}
```


4. 零拷贝数据传输

Datanode 最重要的功能之一就是读取数据块，这个操作看似简单，但在操作系统层面却需要 4 个步骤才能完成。如图 4-45 所示，Datanode 会首先将数据块从磁盘存储（也可能是 SSD、内存等异构存储）读入操作系统的内核缓冲区（步骤 1），再将数据跨内核推到 Datanode 进程（步骤 2），然后 Datanode 会再次跨内核将数据推回内核中的套接字缓冲区（步骤 3），最后将数据写入网卡缓冲区（步骤 4）。可以看到，Datanode 对数据进行了两次多余的数据拷贝操作（步骤 2 和步骤 3），Datanode 只是起到缓存数据并将其传回套接字的作用而以，别无他用。这里需要注意的是，步骤 1 和步骤 4 的拷贝发生在外设（例如磁盘和网卡）和内存之间，由 DMA（Direct Memory Access，直接内存存取）引擎执行，而步骤 2 和步骤 3 的拷贝则发生在内存中，由 CPU 执行。

注：操作系统之所以引入内核缓冲区，是为了提高读写性能。在读操作中，如果应用程序所需的数据量小于内核缓冲区大小时，内核缓冲区可以预读取部分数据，从而提高应用程序的读效率。在写操作中，引入中间缓冲区则可以让写入过程异步完成。

而对于 Datanode，由于读取的数据块文件往往比较大，引入中间缓冲区可能成为一个性能瓶颈，造成数据在磁盘、内核缓冲区和用户缓冲区中被拷贝多次。

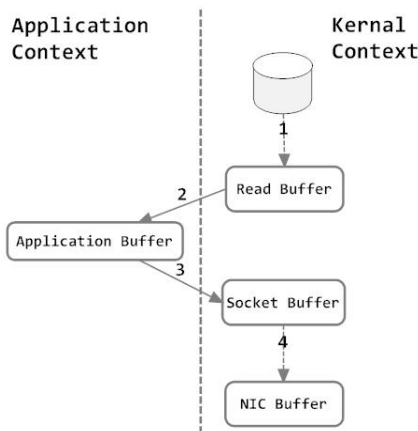


图 4-45 读取数据块时缓冲区拷贝流程图

上述读取方式除了会造成多次数据拷贝操作外，还会增加内核态与用户态之间的上下文切换。如图 4-46 所示，Datanode 通过 `read()` 系统调用将数据块从磁盘（或者其他异构存储）读取到内核缓冲区时，会造成第一次用户态到内核态的上下文切换（切换 1）。之后在系统调用 `read()` 返回时，会触发内核态到用户态的上下文切换（切换 2）。Datanode 成功读入数据后，会调用系统调用 `send()` 发送数据到套接字，也就是在数据块第三次拷贝时，会再次触发用户态到内核态的上下文切换（切换 3）。当系统调用 `send()` 返回时，内核态又会重新切换回用户态。所以这个简单的读取操作，会造成 4 次用户态与内核态之间的上下文切换。

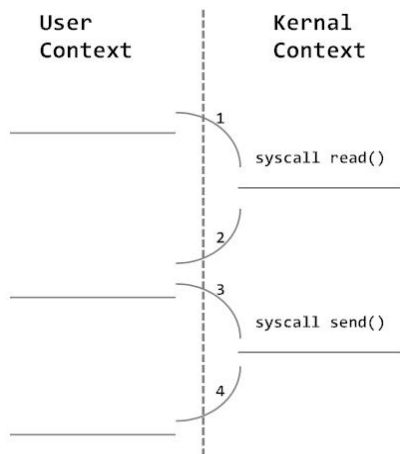


图 4-46 读取数据块的上下文切换示意图

幸运的是，Java NIO 提供了零拷贝模式来消除这些多余的拷贝操作，并且减少内核态与用户态之间的上下文切换。使用零拷贝的应用程序可以要求内核直接将数据从磁盘文件拷贝到网卡缓冲区，而无须通过应用程序周转，从而大大提高了应用程序的性能。Java 类库定义了 `java.nio.channels.FileChannel.transferTo()` 方法，用于在 Linux（UNIX）系统上支持零拷贝，它的声明如下：

```
public abstract long transferTo(long position, long count,
                               WritableByteChannel target) throws IOException;
```

`transferTo()` 方法读取文件通道（`FileChannel`）中 `position` 参数指定位置处开始的 `count` 个字节的数据，然后将这些数据直接写入目标通道 `target` 中。HDFS 的 `SocketOutputStream` 对象的 `transferToFully()` 方法封装了 `FileChannel.transferTo()` 方法，对 `Datanode` 提供支持零拷贝的数据读取功能。`transferToFully()` 方法的定义如下：

```
public void transferToFully(FileChannel fileCh, long position, int count,
                           LongWritable waitWritableTime,
                           LongWritable transferToTime) throws IOException {
```

图 4-47 给出了使用零拷贝读取数据块时缓冲区拷贝流程，`Datanode` 调用 `transferTo()` 方法引发 DMA 引擎将文件内容拷贝到内核缓冲区（步骤 1）。之后数据并未被拷贝到 `Datanode` 进程中，而是由 DMA 引擎直接把数据从内核缓冲区传输到网卡缓冲区（步骤 2）。可以看到，使用零拷贝模式的数据块读取，数据拷贝的次数从 4 次降低到了 2 次。

注：零拷贝模式要求底层网络接口卡支持收集操作，在 Linux 内核 2.4 及后期版本中，套接字缓冲区描述符做了相应调整，可以满足该需求。

使用零拷贝模式除了降低数据拷贝的次数外，上下文切换次数也从 4 次降低到了 2 次。如图 4-48 所示，当 `Datanode` 调用 `transferTo()` 方法时会发生用户态到内核态的切换，`transferTo()` 方法执行完毕返回时内核态又会切换回用户态。

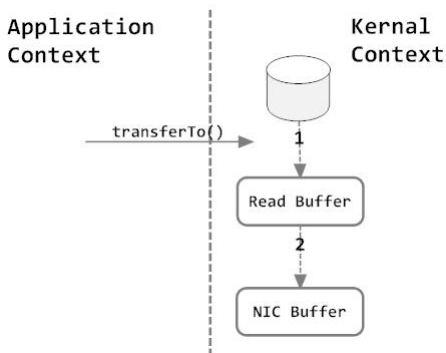


图 4-47 零拷贝模式下缓冲区拷贝流程图

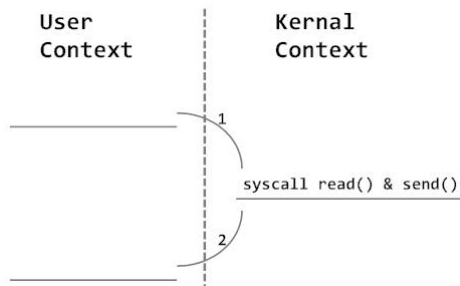


图 4-48 零拷贝模式下上下文切换示意图

BlockSender 使用 `SocketOutputStream.transferToFully()` 封装的零拷贝模式发送数据块到客户端，由于数据不再经过 Datanode 中转，而是直接在内核中完成了数据的读取与发送，所以大大地提高了读取效率。但这也带来了一个问题，由于数据不经过 Datanode 的内存，所以 Datanode 失去了在客户端读取数据块过程中对数据校验的能力。为了解决这个问题，HDFS 将数据块读取操作中的数据校验工作放在客户端执行，客户端完成校验工作后，会将校验结果发送回 Datanode。在 `DataXceiver.readBlock()` 的清理动作中，数据节点会接收客户端的响应码，以获取客户端的校验结果。

5. 节流器

Datanode 是一个典型的 IO 密集型应用，所以 Datanode 的磁盘 I/O 和网络 I/O 往往会成为系统的瓶颈。数据节点上会有多种不同的任务共同占用网络 I/O 和磁盘 I/O，例如数据块读操作、数据块写操作、数据块复制操作以及数据块扫描操作等。如果不对这些操作使用磁盘和网络进行一定的速度控制，那么 Datanode 的吞吐量、延迟以及服务的可用性会大大降低。

所以 Datanode 使用节流器类 `DataTransferThrottler` 来提供简单的流控机制，`DataTransferThrottler` 是一个线程安全类，它可以在 Datanode 上运行的多个任务线程间共享，它的 `bandwidth` 属性描述了所有线程共享的带宽大小。`DataTransferThrottler` 类的使用非常简单，只需要在成功构造对象后，调用 `throttle()` 方法即可。`throttle()` 方法中的 `numOfBytes` 参数指明了当前线程要发送/接收数据的大小，如果 `DataTransferThrottler` 判断 Datanode 上有足够的带宽可以满足请求量，则 `throttle()` 方法直接返回，工作线程就可以立即执行发送/接收数据的操作；否则，`throttle()` 方法会一直阻塞，直到当前请求量得到满足，对应的工作线程也会在 `throttle()` 方法上等待，这样也就达到了流控的作用。`DataTransferThrottler` 还有一个子类 `BlockBalanceThrottler`，这个类是专门为平衡器 (Balancer) 设计的，它除了能够控制 I/O 速度外，还提供了限制使用共享节流器实例的功能。也就是说，`BlockBalanceThrottler` 可以限制同一个 Datanode 上平衡器最多能拥有工作任务的数量。

如图 4-49 所示, `DataTransferThrottler` 的实现也很简单, `period` (描述一个发送周期的时长, 单位是毫秒) 和 `bytesPerPeriod` 字段 (描述每个发送周期所能发送数据的大小) 共同决定了节流器的共享带宽 `bandwidth` (通过 $\text{bandwidth} = \text{bytesPerPeriod} * 1000 / \text{period}$ 公式), `curReserve` 字段则保存着当前发送周期还能发送的剩余数据大小。`DataTransferThrottler.throttle()` 方法会循环判断请求发送数据量是否比剩余数据量小, 如果 `throttle()` 方法能满足请求量则立即返回, 调用线程就可以立即执行发送/接收数据的操作; 否则, `DataTransferThrottler.throttle()` 方法会循环等待检查周期结束, 并在每个周期结束时增加剩余请求量 (每个周期添加 `bytesPerPeriod`), 直到当前请求量得到满足时 `throttle()` 方法才会退出。`throttle()` 方法的代码如下:

```
public synchronized void throttle(long numBytes, Canceler canceler) {

    curReserve -= numBytes;
    bytesAlreadyUsed += numBytes;

    while (curReserve <= 0) {
        if (canceler != null && canceler.isCancelled()) {
            return;
        }
        long now = monotonicNow();
        long curPeriodEnd = curPeriodStart + period;

        if ( now < curPeriodEnd ) {
            // 等待至下一个周期, curReserve 可以增加
            try {
                wait( curPeriodEnd - now );
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        } else if ( now < (curPeriodStart + periodExtension)) {
            curPeriodStart = curPeriodEnd;
            curReserve += bytesPerPeriod; // 增加剩余请求量
        } else {
            curPeriodStart = now; // 长时间没有使用节流器, 重置节流器
            curReserve = bytesPerPeriod - bytesAlreadyUsed;
        }
    }

    bytesAlreadyUsed -= numBytes;
}
```

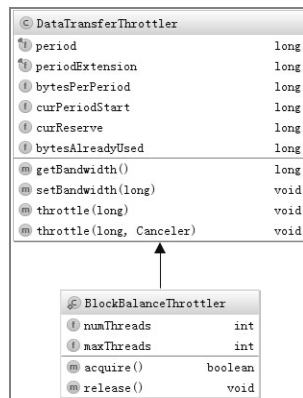


图 4-49 DataTransferThrottler 结构

4.5.6 写数据（done）

流式接口的另一个重要功能是向数据节点写数据，`DataTransferProtocol.write()`方法给出了写操作的接口定义，操作码是 80，`DataXceiver.writeBlock()`则真正实现了 `DataTransferProtocol.writeBlock()`方法。

如图 4-50 所示，HDFS 使用数据流管道方式来写数据。DFSCliet 通过调用 `Sender.writeBlock()`方法触发一个写数据块请求，这个请求会传送到数据流管道中的每一个数据节点，数据流管道中的最后一个数据节点会回复请求确认，这个确认消息逆向地通过数据流管道送回 DFSCliet。DFSCliet 收到请求确认后，将要写入的数据块切分成若干个数据包（packet），然后依次向数据流管道中发送这些数据包。数据包会首先从 DFSCliet 发送到数据流管道中的第一个数据节点（这里是 Datanode1），Datanode1 成功接收数据包后，会将数据包写入磁盘，然后将数据包发送到数据流管道中的第二个节点（Datanode2）。依此类推，当数据包到达数据流管道中的最后一个节点（Datanode3）时，Datanode3 会对收到的数据包进行校验，如果校验成功，Datanode3 会发送数据包确认消息，这个确认消息会逆向地通过数据流管道送回 DFSCliet。当一个数据块中的所有数据包都成功发送完毕，并且收到确认消息后，DFSCliet 会发送一个空数据包标识当前数据块发送完毕。至此，整个数据块发送流程结束。

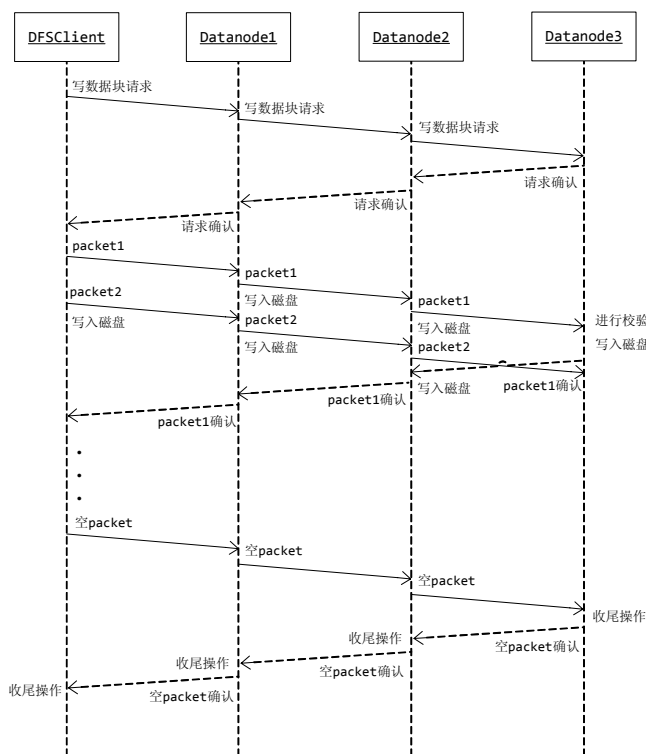


图 4-50 以数据流管道方式写数据流程图

如图 4-51 所示, 这个写数据块请求通过流式接口到达 Datanode 之后, Datanode 上监听流式接口请求的 DataXceiverServer 会接收这个请求, 并构造一个 DataXceiver 对象, 然后在 DataXceiver 对象上调用 DataXceiver.writeBlock() 方法响应这个请求。当前 Datanode 的 DataXceiver.writeBlock() 方法会级联向数据流管道中的下一个 Datanode 发送写数据块请求, 这个流式请求会一直在数据流管道中传递下去, 直到写数据块请求到达数据流管道中的最后一个 Datanode。

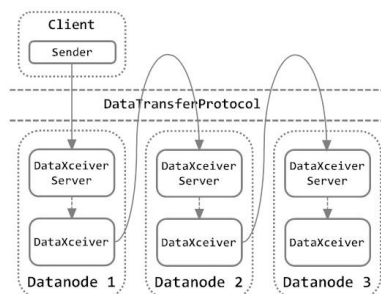


图 4-51 数据流管道中的流式接口请求传递示意图

1. DataXceiver.writeBlock() (done)

我们首先看一下 DataXceiver.writeBlock() 方法的实现。DataXceiver.writeBlock() 方法实现了向数据节点写数据的所有逻辑, writeBlock() 的处理逻辑非常复杂, 这里先给出 writeBlock() 方法操作流程图, 如图 4-52 所示, 我们将按照这个流程图介绍 writeBlock() 方法的实现。

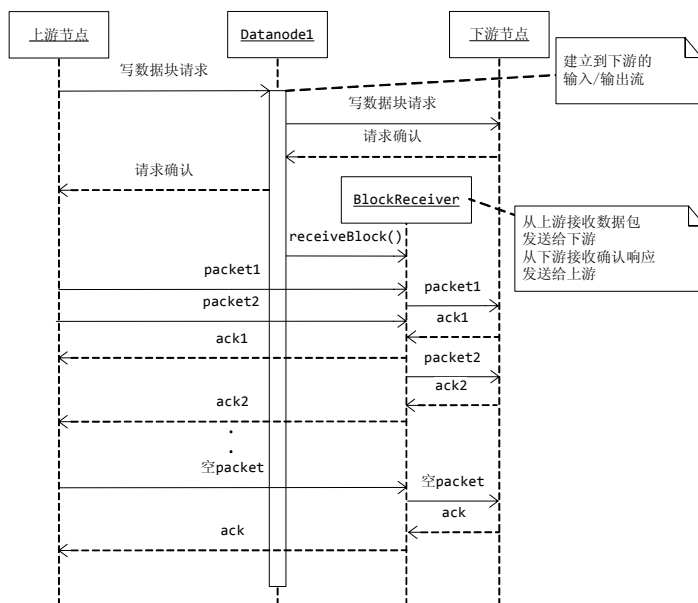


图 4-52 writeBlock() 方法操作流程图

`writeBlock()`方法的开始部分定义了三个 `boolean` 类型的变量，用于控制处理流程。我们先看一下这三个变量的作用。

```
// isDatanode 变量指示当前写操作是否是 DFSClient 发起的
final boolean isDatanode = clientname.length() == 0;
// isClient 变量与 isDatanode 相反，表示是 Datanode 触发的写操作
final boolean isClient = !isDatanode;
// isTransfer 变量指示当前的写操作是否为数据块复制操作，利用数据流管道状态来判断
final boolean isTransfer = stage == BlockConstructionStage.TRANSFER_RBW
    || stage == BlockConstructionStage.TRANSFER_FINALIZED;
```

对于客户端发起的写数据块请求，这里三个变量的值分别为 `isDatanode`—`false`，`isClient`—`true`，`isTransfer`—`false`，本节的讲解也都是按照这个初始值进行的。

在 `writeBlock()`方法中还使用到了两组输入/输出流，如图 4-53 所示。

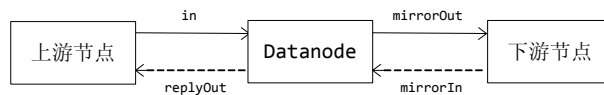


图 4-53 数据流管道的两组输入/输出流

`Datanode` 与数据流管道中的上游节点通信用到了输入流 `in` 以及输出流 `replyOut`，与数据流管道中的下游节点通信则用到了输入流 `mirrorIn` 以及输出流 `mirrorOut`。`writeBlock()`方法的第二部分就是初始化这两组输入/输出流，并向下游节点发送数据包写入请求，然后等待下游节点的请求确认。如果下游节点确认了请求，则向上游节点返回这个确认请求；如果抛出了异常，则向上游节点发送异常响应。代码如下：

```
// 创建 replyOut 输出流
final DataOutputStream replyOut = new DataOutputStream(
    new BufferedOutputStream(
        getOutputStream(),
        HdfsConstants.SMALL_BUFFER_SIZE));

DataOutputStream mirrorOut = null; // 到下游数据节点的输出流
DataInputStream mirrorIn = null;  // 下游数据节点的输入流
Socket mirrorSock = null;         // 到下游节点的 Socket
String mirrorNode = null;         // 下游节点的名称：端口
String firstBadLink = "";         // 数据流管道中第一个失败的 Datanode
Status mirrorInStatus = SUCCESS;
final String storageUuid;         // 保存这个数据块的 Datanode 存储的 id
try {
    // 打开一个 BlockReceiver，用于从上游节点接收数据块
    blockReceiver = new BlockReceiver(block, storageType, in,
        peer.getRemoteAddressString(),
        peer.getLocalAddressString(),
        stage, latestGenerationStamp, minBytesRcvd, maxBytesRcvd,
        clientname, srcDataNode, datanode, requestedChecksum,
```

Hadoop 2.X HDFS 源码剖析

```
        cachingStrategy, allowLazyPersist));

// 连接到下游节点
if (targets.length > 0) {
    mirrorNode = targets[0].getXferAddr(connectToDnViaHostname);
    mirrorTarget = NetUtils.createSocketAddr(mirrorNode);
    mirrorSock = datanode.newSocket();
    try {
        // 建立到下游节点的 Socket 连接
        NetUtils.connect(mirrorSock, mirrorTarget, timeoutValue);
        mirrorSock.setSoTimeout(timeoutValue);
        mirrorSock.setSendBufferSize(HdfsConstants.DEFAULT_DATA_SOCKET_SIZE);
        // 创建 mirrorOut 和 mirrorIn 建立到下游节点的输出流以及输入流
        mirrorOut = new DataOutputStream(new BufferedOutputStream(unbufMirrorOut,
            HdfsConstants.SMALL_BUFFER_SIZE));
        mirrorIn = new DataInputStream(unbufMirrorIn);

        // 向下游节点发送数据块写入请求
        new Sender(mirrorOut).writeBlock(originalBlock, targetStorageTypes[0],
            blockToken, clientname, targets, targetStorageTypes, srcDataNode,
            stage, pipelineSize, minBytesRcvd, maxBytesRcvd,
            latestGenerationStamp, requestedChecksum, cachingStrategy, false);
        mirrorOut.flush();
        // 接收来自下游节点的请求确认, 并记录请求确认状态
        BlockOpResponseProto connectAck =
            BlockOpResponseProto.parseFrom(PBHelper.vintPrefixed(mirrorIn));
        mirrorInStatus = connectAck.getStatus();
        firstBadLink = connectAck.getFirstBadLink();
    } catch (IOException e) {
        // 出现异常, 向上游节点发送异常响应
        if (isClient) {
            BlockOpResponseProto.newBuilder()
                .setStatus(ERROR)
                .setFirstBadLink(targets[0].getXferAddr())
                .build()
                .writeDelimitedTo(replyOut);
            replyOut.flush();
        }
        // 关闭到下游节点的 Socket、输入流以及输出流
        IOUtils.closeStream(mirrorOut);
        mirrorOut = null;
        IOUtils.closeStream(mirrorIn);
        mirrorIn = null;
        IOUtils.closeSocket(mirrorSock);
        mirrorSock = null;
        if (isClient) {
            throw e;
        } else {

```



```

    }
  }
}

if (isClient && !isTransfer) {
  // 向上游节点返回请求确认
  BlockOpResponseProto.newBuilder()
    .setStatus(mirrorInStatus)
    .setFirstBadLink(firstBadLink)
    .build()
    .writeDelimitedTo(replyOut);
  replyOut.flush();
}

```

成功建立了与下游节点的输入/输出流后，writeBlock()方法就会调用 blockReceiver.receiveBlock()方法从上游节点接收数据块，然后将数据块发送到下游节点。同时 blockReceiver 对象还会从下游节点接收数据块中数据包的确认消息，并且将这个确认消息转发到上游节点。

```

if (blockReceiver != null) {
  String mirrorAddr = (mirrorSock == null) ? null : mirrorNode;
  // 调用 BlockReceiver.receiveBlock() 从上游节点接收数据块，然后将数据块发送到下游节点
  blockReceiver.receiveBlock(mirrorOut, mirrorIn, replyOut,
    mirrorAddr, null, targets, false);

  // 对于复制操作，不需要向下游节点转发数据块，也不需要接收下游节点的确认
  // 所以成功接收完数据块之后，在当前节点直接返回确认消息
  if (isTransfer) {
    writeResponse(SUCCESS, null, replyOut);
  }
}

```

成功执行了 BlockReceiver.receiveBlock()之后，writeBlock()方法就会更新当前数据节点上新写入数据块副本的时间戳、副本文件长度等信息。如果是数据流管道关闭的恢复操作或者是数据块的复制操作，则调用 Datanode.closeBlock()方法向 Namenode 汇报 Datanode 接收了新的数据块，Datanode.closeBlock()调用 BPOfferService.notifyNamenodeReceivedBlock()通知 Namenode。对于客户端发起的写数据请求，在 PacketResponder 线程中已经通过调用 Datanode.closeBlock()方法关闭了数据块（请参考 BlockReceiver.PacketResponder 小节）。

```

if (isClient &&
    stage == BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {
  block.setGenerationStamp(latestGenerationStamp);
  block.setNumBytes(minBytesRcvd);
}

if (isDatanode ||
    stage == BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {
  datanode.closeBlock(block, DataNode.EMPTY_DEL_HINT, storageUuid);
}

```

`DataXceiver.writeBlock()`完成上述操作后,会在 `finally` 中关闭上下游节点的输入/输出流,同时关闭 `blockReceiver` 对象。至此,一个完整的写数据流程就结束了。

```

} finally {
    // close all opened streams
    IOUtils.closeStream(mirrorOut);
    IOUtils.closeStream(mirrorIn);
    IOUtils.closeStream(replyOut);
    IOUtils.closeSocket(mirrorSock);
    IOUtils.closeStream(blockReceiver);
    blockReceiver = null;
}

```

2. BlockReceiver (done)

`BlockReceiver` 类负责从数据流管道中的上游节点接收数据块,然后保存数据块到当前数据节点的存储中,再将数据块转发到数据流管道中的下游节点。同时 `BlockReceiver` 还会接收来自下游节点的响应,并把这个响应发送给数据流管道中的上游节点。`BlockReceiver` 接收数据块的流程如图 4-54 所示。

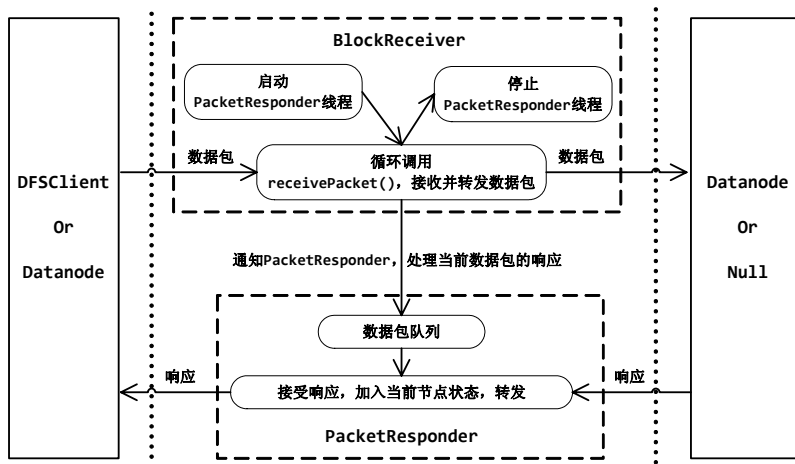


图 4-54 BlockReceiver 接收数据块流程图

`BlockReceiver.receiveBlock()`方法的逻辑比较简单,它先启动 `PacketResponder` 线程负责接收并转发下游节点发送的确认数据包的 ACK 消息。之后 `receiveBlock()`方法循环调用 `receivePacket()`方法接收上游写入的数据包并转发这个数据包到下游节点。成功完成整个数据块的写入操作后, `receiveBlock()`方法关闭 `PacketResponder` 线程。

```

void receiveBlock() {
    // 启动 PacketResponder 线程处理确认包的接收和转发
    responder = new Daemon(datanode.threadGroup,
        new PacketResponder(replyOut, mirrorIn, downstreams));
    responder.start();
}

```

```
// 循环调用 receivePacket() 接收并转发数据块中的所有数据包
while (receivePacket() >= 0) {}

// 完成数据块的写入操作后，结束 PacketResponder 线程
((PacketResponder) responder.getRunnable()).close();
responderClosed = true;
}
```

可以看到，receiveBlock()方法将大部分处理工作都交给了 receivePacket()方法，receivePacket()方法的处理流程如图 4-55 所示。

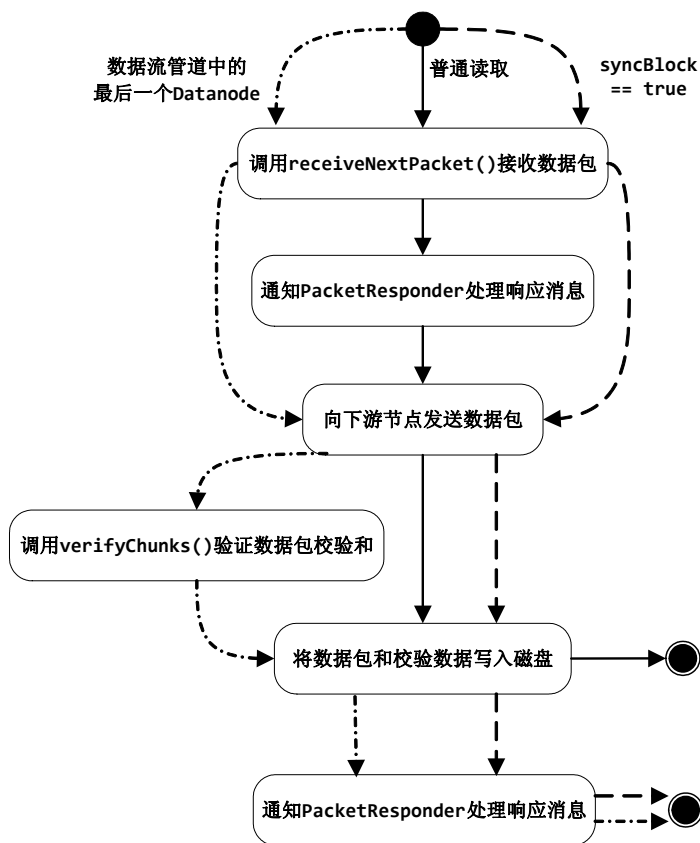


图 4-55 receivePacket()方法处理流程图

receivePacket()方法首先调用 packetReceiver.receiveNextPacket()方法从输入流中读入一个数据包 (packet)，并将这个数据包放入 ByteBuffer 缓冲区 curPacketBuf 中。readNextPacket()方法的实现比较简单，就是按照数据包格式从输入流中读取数据并放入指定的 ByteBuffer 缓冲区中。数据包格式的定义请参考读数据的数据块的传输格式小节。

`receivePacket()`成功接收数据包后,会判断当前节点是否是数据流管道中的最后一个节点,或者是输入流启动了 `sync` 标识 (`syncBlock`) 要求 `Datanode` 立即将数据包同步到磁盘。在这两种情况下, `Datanode` 会先将数据写入磁盘,然后再通知 `PacketResponder` 处理确认 (ACK) 消息; 否则, `receivePacket()`方法接收完数据包后会立即通知 `PacketResponder` 处理确认消息。

接下来 `receivePacket()`会将数据包发送给数据流管道中的下游节点,然后就可以将数据块文件和校验文件写入数据节点的磁盘了。写入磁盘之后, `receivePacket()`方法需要调用 `flushOrSync()` 方法将输出流缓存中的数据全部同步到磁盘,最后还需要调用 `manageWriterOsCache` (清理) 操作系统缓存中的数据。需要注意的是,如果当前节点是数据流管道中的最后一个节点,则在写入磁盘前,需要先对数据块中的所有数据包进行校验。

由于在 `Datanode` 是数据流管道中最后一个节点,以及携带了 `sync` 标识两种情况下, `receivePacket()`方法并没有通知 `PacketResponder` 处理响应消息,所以在 `receiverPacket()`方法的最后通知 `PacketResponder` 为这两种情况处理响应。

```
private int receivePacket() throws IOException {
    // 调用 packetReceiver.receiveNextPacket() 方法从输入流中读入一个数据包
    packetReceiver.receiveNextPacket(in);
    // ...
    // 如果不是数据流管道中的最后一个数据节点,则立即处理响应消息
    if (responder != null && !syncBlock && !shouldVerifyChecksum()) {
        ((PacketResponder) responder.getRunnable()).enqueue(seqno,
            lastPacketInBlock, offsetInBlock, Status.SUCCESS);
    }

    // 向下游节点发送数据包
    if (mirrorOut != null && !mirrorError) {
        packetReceiver.mirrorPacketTo(mirrorOut);
        mirrorOut.flush();
    }

    if (lastPacketInBlock || len == 0) {
        // 如果接收了完整的数据块,并且启动了 sync 标识,则立即将数据同步到磁盘
        if (syncBlock) {
            flushOrSync(true);
        }
    } else {
        // 如果当前节点是数据流管道中的最后一个节点,则验证数据包的校验和
        if (checksumReceivedLen > 0 && shouldVerifyChecksum()) {
            try {
                // 调用 verifyChunks() 验证数据包校验和
                verifyChunks(dataBuf, checksumBuf);
            } catch (IOException ioe) {
                // 验证出现异常,则向上游客户端发送校验异常消息
                if (responder != null) {
                    try {
                        ((PacketResponder) responder.getRunnable()).enqueue(seqno,
```

```

        lastPacketInBlock, offsetInBlock,
        Status.ERROR_CHECKSUM);

        Thread.sleep(3000);
    } catch (InterruptedException e) { }
}
throw new IOException("Terminating due to a checksum error." + ioe);
}
if (needsChecksumTranslation) {
    // 如果客户端发送的数据校验方式和当前数据节点的不一致，则转换校验和
    translateChunks(dataBuf, checksumBuf);
}
}

// 写入数据和校验数据
out.write(...);
checksumOut.write(...);
// 将数据和校验数据同步到磁盘
flushOrSync(syncBlock);
// 清除操作系统缓存
manageWriterOsCache(offsetInBlock);
}

// 如果是最后一个节点，或者是 syncBlock 方式，则在数据写完磁盘以后再对 ack 进行处理
if (responder != null && (syncBlock || shouldVerifyChecksum())) {
    ((PacketResponder) responder.getRunnable()).enqueue(seqno,
        lastPacketInBlock, offsetInBlock, Status.SUCCESS);
}

// 节流器控制
if (throttler != null) { // throttle I/O
    throttler.throttle(len);
}
}
}

```

3. BlockReceiver.PacketResponder

通过前面小节的介绍我们知道，BlockReceiver 负责从上游数据节点接收数据包并转发到下游数据节点。同时，如果当前节点是数据流管道中的最后一个节点，BlockReceiver 还需要验证数据包的校验和是否正确。除此之外，数据节点还需要从下游节点接收数据包的确认消息，然后转发给上游节点，BlockReceiver 将这部分功能委托给了内部类 PacketResponder。PacketResponder 是一个独立的线程类，它和 BlockReceiver 所在的线程共同完成数据块的写操作流程。这里之所以将数据块处理和数据块响应消息处理放在两个线程中，是因为如果使用一个线程处理，需要同时监听上游节点的输入流和下游节点的输入流，任意一个输入流阻塞都会造成另一个输入流上消息处理的延迟。

本节就介绍 PacketResponder 线程类的实现，它的处理流程如图 4-56 所示。BlockReceiver

完成对指定数据包的处理之后，会触发 `PacketResponder` 类处理当前数据包的响应消息。`PacketResponder` 类监听下游的输入流，接收到这个数据包的确认消息后，在确认消息中添加当前数据节点的消息，然后将这个消息发送给上游数据节点。

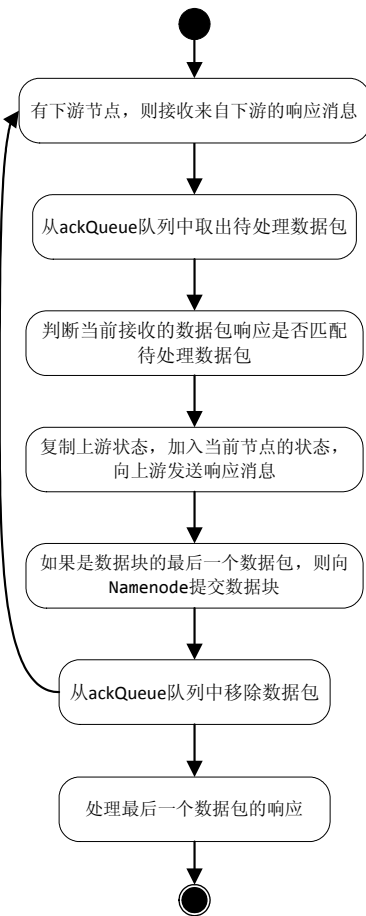


图 4-56 `PacketResponder` 处理流程图

我们首先看一下 `PacketResponder.enqueue()` 方法。`BlockReceiver` 完成对指定数据包的处理之后，会调用 `enqueue()` 方法通知 `PacketResponder` 类处理这个数据包的响应。`enqueue()` 方法的实现比较简单，它将等待 `PacketResponder` 类处理的数据包加入 `ackQueue` 队列中，`ackQueue` 队列中保存的所有数据包都会由 `PacketResponder` 的 `run()` 方法处理。成功将数据包添加到 `ackQueue` 队列后，`enqueue()` 方法会调用 `notify()` 方法通知 `run()` 方法处理数据包。可以看到，`ackQueue` 是一个典型的生产者-消费者队列。

```

void enqueue(final long seqno, final boolean lastPacketInBlock,
             final long offsetInBlock, final Status ackStatus) {
    final Packet p = new Packet(seqno, lastPacketInBlock, offsetInBlock,

```

```

        System.nanoTime(), ackStatus);
    synchronized(ackQueue) {
        if (running) {
            ackQueue.addLast(p);
            ackQueue.notifyAll();
        }
    }
}

```

我们看一下 `PacketResponder.run()` 方法的实现。`run()` 方法会循环对数据块中的所有数据包执行确认响应的逻辑，如果抛出异常，则将 `PacketResponder.running` 设置为 `false`，使得 `while` 循环的 `isRunning()` 判断为 `false`，`PacketResponder` 线程也就会结束。如果 `PacketResponder` 阻塞了，则通过 `interrupt()` 方法中断 `PacketResponder` 线程。`run()` 方法执行数据包响应的逻辑比较复杂，方法也很长，我们分为几块来学习。

```

public void run() {
    boolean lastPacketInBlock = false;
    while (isRunning() && !lastPacketInBlock) {
        try {
            // 执行数据包响应处理逻辑
        } catch (IOException e) {
            if (running) {
                // 检测当前节点上的数据是否有错
                datanode.checkDiskErrorAsync();
                // 设置 running 标志位为 false，停止 PacketResponder 线程的执行
                running = false;
                if (!Thread.interrupted()) {
                    // 中断 PacketResponder 线程
                    receiverThread.interrupt();
                }
            }
        } catch (Throwable e) {
            if (running) {
                running = false; // 设置 running 标志位为 false
                receiverThread.interrupt(); // 中断 PacketResponder 线程
            }
        }
    }
}

```

`run()` 方法首先会从下游节点的输入流中读取一个响应，并判断这个响应中是否有 OOB 消息（`Datanode` 在写操作时被触发重启的情况下，会通过数据流管道逆向发送一个 OOB 响应消息给客户端，由客户端处理数据流管道中 `Datanode` 节点重启的情况），如果有 OOB 消息，则立即将这个消息转发给上游数据节点。接下来 `run()` 方法会在 `ackQueue` 队列上等待需要处理的数据包，然后判断从下游节点接收的数据包响应与从 `ackQueue` 队列中取出的待处理数据包是否匹配，如果不匹配则抛出异常。这里需要注意，如果 `PacketResponder` 在从下游节点读入 `ack` 的过程中出现异常，则将 `mirrorError` 字段设置为 `true`，`run()` 方法会在后续向上游节点发送的响应中携带错误信息。

Hadoop 2.X HDFS 源码剖析

```
Packet pkt = null; // 记录当前处理的数据包
long expected = -2; // 数据包的序列化
PipelineAck ack = new PipelineAck(); // 数据包响应消息
long seqno = PipelineAck.UNKOWN_SEQNO; // 数据包响应的序列号
long ackRecvNanoTime = 0;
try {
    if (type != PacketResponderType.LAST_IN_PIPELINE && !mirrorError) {
        // 从下游节点的输出流中读入一个响应
        ack.readFields(downstreamIn);
        // 判断响应消息是否是 OOB 消息, 例如下游节点重启
        Status oobStatus = ack.getOOBStatus();
        if (oobStatus != null) {
            // 将 OOB 消息转发给上游节点处理
            sendAckUpstream(ack, PipelineAck.UNKOWN_SEQNO, 0L, 0L,
                Status.SUCCESS);
            continue;
        }
        seqno = ack.getSeqno();
    }
    if (seqno != PipelineAck.UNKOWN_SEQNO
        || type == PacketResponderType.LAST_IN_PIPELINE) {
        // 从 ackQueue 队列中取出待处理数据包
        pkt = waitForAckHead(seqno);
        if (!isRunning()) {
            break;
        }
        expected = pkt.seqno;
        // 判断当前接收的数据包响应是否匹配待处理数据包
        if (type == PacketResponderType.HAS_DOWNSTREAM_IN_PIPELINE
            && seqno != expected) {
            // 如果不匹配则抛出异常
            throw new IOException(myString + "seqno: expected=" + expected
                + ", received=" + seqno);
        }
        lastPacketInBlock = pkt.lastPacketInBlock;
    }
} catch (InterruptedException ine) {
    isInterrupted = true;
} catch (IOException ioe) {
    if (Thread.interrupted()) {
        isInterrupted = true;
    } else {
        // 如果从下游节点读取数据时抛出异常, 则将 mirrorError 设置为 true
        mirrorError = true;
    }
}
if (Thread.interrupted() || isInterrupted) {
    // 对于线程中断, 则将 running 设置为 false, 停止当前线程运行
```



```

    running = false;
    continue;
}

```

完成上述操作后，PacketResponder 会判断当前接收的数据包响应是否为数据块中最后一个数据包的响应，如果是，则调用 finalizeBlock() 方法向 Namenode 提交这个数据块。接下来 run() 方法会调用 sendAckUpstream() 方法复制下游节点的数据包响应，并在该响应中加入当前节点的状态，然后构造新的响应发送给上游节点。完成数据包响应的处理后，从 ackQueue 队列中移除这个数据包。

```

if (lastPacketInBlock) {
    // 如果数据包响应是数据块的最后一个数据包响应，则提交这个数据块
    finalizeBlock(startTime);
}
// 复制下游节点的数据包响应，加入当前数据节点的状态，并发送给上游节点
sendAckUpstream(ack, expected, totalAckTimeNanos,
    (pkt != null ? pkt.offsetInBlock : 0),
    (pkt != null ? pkt.ackStatus : Status.SUCCESS));
if (pkt != null) {
    // 已经完成响应处理的数据包，从 ackQueue 队列中移除
    removeAckHead();
}

```

这里我们看一下 sendAckUpstream() 方法的实现。sendAckUpstream() 方法会调用 sendAckUpstreamUnprotected() 方法发送响应给上游节点。向上游节点发送的响应分为如下几种情况。

- 下游节点发送的是 OOB 消息，则将下游节点的数据包响应原封不动地保存。
- 如果从下游节点读取响应异常，也就是 mirrorError 为 true，则将异常 Status.MIRROR_ERROR_STATUS 记录在数据包响应中。
- 对于其他流程，则将当前 Datanode 的状态放入数据包响应中。

然后构造新的数据包响应，并将这个响应发送到上游数据节点。sendAckUpstream() 方法的代码如下：

```

private void sendAckUpstreamUnprotected(PipelineAck ack, long seqno,
    long totalAckTimeNanos, long offsetInBlock, Status myStatus)
    throws IOException {
    Status[] replies = null;
    if (ack == null) { // 如果下游节点发送的是 OOB 消息，则保留下游节点消息内容
        replies = new Status[1];
        replies[0] = myStatus;
    } else if (mirrorError) { // 如果是当前节点从下游节点读取响应消息异常
        // 则将当前 Datanode 的错误记录在响应消息中
        replies = MIRROR_ERROR_STATUS;
    } else { // 其他流程
        short ackLen = type == PacketResponderType.LAST_IN_PIPELINE ? 0 : ack
            .getNumOfReplies();
    }
}

```

```
replies = new Status[1 + ackLen];
replies[0] = myStatus; // 将当前数据的状态放入响应中
for (int i = 0; i < ackLen; i++) { // 将下游数据节点的状态放入响应中
    replies[i + 1] = ack.getReply(i);
}
// 如果上游节点检测到校验和错误, 那么当前节点发送的数据有错误
// 则停止 BlockReceiver 以及 PacketResponder 线程
if (ackLen > 0 && replies[1] == Status.ERROR_CHECKSUM) {
    throw new IOException("Shutting down writer and responder "
        + "since the down streams reported the data sent by this "
        + "thread is corrupt");
}
}

// 构造新的数据包响应消息
PipelineAck replyAck = new PipelineAck(seqno, replies,
    totalAckTimeNanos);
if (replyAck.isSuccess()
    && offsetInBlock > replicaInfo.getBytesAked()) {
    replicaInfo.setBytesAked(offsetInBlock);
}
// 将数据包响应消息发送给上游节点
long begin = Time.monotonicNow();
replyAck.write(upstreamOut);
upstreamOut.flush();

// 如果当前节点检测到校验和错误, 则停止 BlockReceiver 以及 PacketResponder 线程
if (myStatus == Status.ERROR_CHECKSUM) {
    throw new IOException("Shutting down writer and responder "
        + "due to a checksum error in received data. The error "
        + "response has been sent upstream.");
}
}
```

这里需要特别注意的是, 如果发现下游节点汇报了数据包的校验和错误, 则抛出异常停止当前节点的 **BlockReceiver** 以及 **PacketResponder** 线程; 如果是当前节点出现了数据包的校验和错误, 则将错误消息发送给上游节点, 然后抛出异常停止当前节点的 **BlockReceiver** 以及 **PacketResponder** 线程。

4. 处理结果上报名字节点

不同于数据块的读取操作, **Datanode** 完成数据块的写入操作后需要向 **Namenode** 汇报这个新的数据块, 以方便 **Namenode** 更新命名空间。**PacketResponder** 在确认数据块中所有数据包的响应都正确处理之后, 会调用 **BlockReceiver.finalizeBlock()** 方法通知名字节点当前 **Datanode** 成功接收了这个数据块。**finalizeBlock()** 方法又会调用 **DataNode.closeBlock()** 方法, **DataNode.closeBlock()** 在底层调用了 **BPOfferService.notifyNamenodeReceivedBlock()** 方法通知 **Namenode**。**BPOfferService** 的 **notifyNamenodeReceivedBlock()** 方法实现我们在 **BPOfferService**

实现小节中已经介绍过了。

```
private void finalizeBlock(long startTime) throws IOException {
    BlockReceiver.this.close();
    final long endTime = ClientTraceLog.isInfoEnabled() ? System.nanoTime()
        : 0;
    block.setNumBytes(replicaInfo.getNumBytes());
    datanode.data.finalizeBlock(block);
    datanode.closeBlock(
        block, DataNode.EMPTY_DEL_HINT, replicaInfo.getStorageUuid());
}
```

在写数据流程中,另一处调用 `DataNode.closeBlock()`方法是在 `DataXceiver.writeBlock()`中,当这个写操作是由数据块复制操作触发,并且 `Datanode` 已经调用 `blockReceiver.receiveBlock()`成功保存了数据块时,会调用这个方法通知名字节点。

```
if (isDatanode ||
    stage == BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {
    datanode.closeBlock(block, DataNode.EMPTY_DEL_HINT, storageUuid);
}
```

4.5.7 数据块替换、数据块拷贝和读数据块校验

在 `DataXceiver` 实现的流式接口中还包括 `replaceBlock()`、`copyBlock()`以及 `blockChecksum()`三个与数据块操作相关的方法。`replaceBlock()`方法用于将某个 `Datanode` 上的数据块移动到另外一个 `Datanode` 上,这个操作主要用在数据块平衡操作 (balancing) 的场景下。`copyBlock()`方法用于在当前 `Datanode` 上复制数据块, `blockChecksum()`方法则用于获取数据块的校验和。这三个方法的实现都比较简单,由于篇幅原因这里就不再介绍了,请读者自行参考源码。

4.5.8 短路读操作

流式接口中的 `requestShortCircuitFds()`、`releaseShortCircuitFds()`以及 `requestShortCircuitShm()`方法都用于支持短路读取功能。这部分内容和客户端读取操作的联系很紧密,我们在第 5 章中会详细介绍,这里先暂时略过。

4.6 数据块扫描器

每个 `Datanode` 都会初始化一个数据块扫描器周期性地验证 `Datanode` 上存储的所有数据块的正确性,并把发现的损坏数据块报告给 `Namenode`。`DataBlockScanner` 类就是 `Datanode` 上数据块扫描器的实现。由于 `Datanode` 会保存多个块池的数据块,所以 `DataBlockScanner` 会持有多个 `BlockPoolSliceScanner` 对象,每个 `BlockPoolSliceScanner` 对象都负责验证一个指定块池下数据块的正确性。

4.6.1 DataBlockScanner 实现

我们首先学习 DataBlockScanner 的实现，然后在下一小节中介绍 BlockPoolSliceScanner 的实现。DataBlockScanner 类结构如图 4-57 所示。

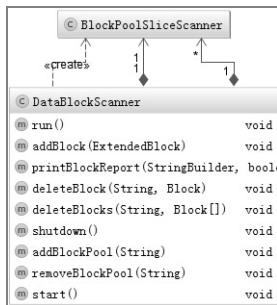


图 4-57 DataBlockScanner 类结构

DataBlockScanner 会通过 blockPoolScannerMap 字段持有所有的 BlockPoolSliceScanner 对象，当 Datanode 新添加或者删除 BPOfferService 对象时，会调用 DataBlockScanner 的 addBlockPool() 以及 removeBlockPool() 方法更新 blockPoolScannerMap 字段。blockPoolScannerMap 的定义代码如下所示。

```
private final TreeMap<String, BlockPoolSliceScanner> blockPoolScannerMap =
    new TreeMap<String, BlockPoolSliceScanner>();
```

当在 Datanode 上新添加或者删除了一个数据块时，需要调用 DataBlockScanner 的 addBlock() 或者 deleteBlock() 方法从 DataBlockScanner 中添加或者移除这些数据块，以保证后续数据块扫描操作的完整性和正确性。例如，在数据块写入流程中，当 BlockReceiver.PacketResponder 确认数据块写入完成后，会调用 DataNode.closeBlock() 方法提交这个数据块，同时还会调用 DataBlockScanner.addBlock() 方法更新 DataBlockScanner 中需要扫描的数据块。

DataBlockScanner 是一个线程类，它会周期性地从所有 blockPoolScannerMap 集合中选出一个最长时间没有进行扫描的 BlockPoolSliceScanner 对象，然后在这个对象上调用 scanBlockPoolSlice() 扫描这个块池下存储的所有数据块。当 run() 方法执行完毕后，会关闭所有的 BlockPoolSliceScanner 对象，完成清理操作。DataBlockScanner.run() 方法的代码如下：

```
public void run() {
    String currentBpId = "";
    boolean firstRun = true;
    while (datanode.shouldRun && !Thread.interrupted()) {
        // 除了第一次迭代外，每次睡眠 5 秒
        if (!firstRun) {
            try {
                Thread.sleep(SLEEP_PERIOD_MS);
            } catch (InterruptedException ex) {
```

```

        blockScannerThread.interrupt();
        continue;
    }
} else {
    firstRun = false;
}

// 调用 getNextBPScanner() 方法选出最长时间没有更新的 BlockPoolSliceScanner
BlockPoolSliceScanner bpScanner = getNextBPScanner(currentBpId);
if (bpScanner == null) {
    continue;
}
currentBpId = bpScanner.getBlockPoolId();
// 如果块池对应的 BPOfferService 对象不再响应, 则移除对应的块池
if (!datanode.isBPServiceAlive(currentBpId)) {
    //调用 removeBlockPool() 方法从 DataBlockScanner.blockPoolScannerMap 中移除
    removeBlockPool(currentBpId);
    continue;
}
// 调用 BlockPoolSliceScanner.scanBlockPoolSlice() 扫描块池下的数据块
bpScanner.scanBlockPoolSlice();
}

// DataBlockScanner 完成执行逻辑后, 会关闭所有的 BlockPoolSliceScanner 对象
for (BlockPoolSliceScanner bpss: blockPoolScannerMap.values()) {
    bpss.shutdown();
}
}

```

通过上面的学习我们知道, 数据块扫描操作实际上是在 `BlockPoolSliceScanner` 对象中执行的。下面我们学习 `BlockPoolSliceScanner` 的实现。

4.6.2 BlockPoolSliceScanner 实现

`BlockPoolSliceScanner` 对象用于验证一个指定块池下的所有数据块, 如图 4-58 所示, `BlockPoolSliceScanner` 定义了内部类 `LogFileHandler`、`BlockScanInfo`、`LogEntry` 以及枚举类 `ScanType` 辅助数据块的验证操作。

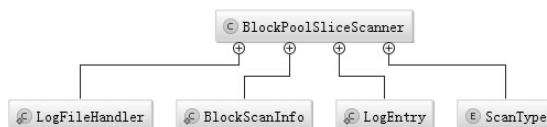


图 4-58 `BlockPoolSliceScanner` 定义的内部类

我们逐一介绍这几个内部类的作用。`BlockScanInfo` 类保存了数据块的扫描信息, 为了节省 `Datanode` 的内存空间, `BlockScanInfo` 被设计为 `Block` 的子类, 而不是持有 `Block` 对象的引

用。这种设计方式在每个数据块副本上可以节省大约 24 字节的内存。**BlockScanInfo** 是可比较的，它提供了内部的比较器对象 **LAST_SCAN_TIME_COMPARATOR**，这个对象会比较 **BlockScanInfo.lastScanTime** 字段的大小，并按时间先后排序。这样，**BlockScanInfo.blockInfoSet** 以及 **BlockScanInfo.newBlockInfoSet** 集合中保存的 **BlockScanInfo** 对象就会按照扫描时间排序了。**BlockScanInfo** 类的定义如下代码所示。

```
static class BlockScanInfo extends Block
    implements LightweightGSet.LinkedElement {

    static final Comparator<BlockScanInfo> LAST_SCAN_TIME_COMPARATOR
        = new Comparator<BlockPoolSliceScanner.BlockScanInfo>() {
        @Override
        public int compare(BlockScanInfo left, BlockScanInfo right) {
            final long l = left.lastScanTime;
            final long r = right.lastScanTime;
            // 比较 lastScanTime 的大小，如果相同则比较 Block
            return l < r? -1: l > r? 1: left.compareTo(right);
        }
    };

    long lastScanTime = 0; // 数据块最后一次扫描时间
    ScanType lastScanType = ScanType.NONE; // 扫描的类型，定义在 ScanType 中
    boolean lastScanOk = true; // 扫描结果，true 表示成功，false 表明失败
    private LinkedElement next; // 连接下一个 BlockScanInfo 对象
    // ...
    BlockScanInfo(Block block) {
        super(block);
    }
    // ...
}
```

BlockScanInfo.lastScanType 字段定义了上一次扫描的类型，这个类型是由内部枚举类 **ScanType** 定义的。**ScanType** 定义了两种扫描类型：**NONE** 用于描述还没有执行过扫描的情况；**VERIFICATION_SCAN** 则表示扫描结果是由数据块扫描器产生的情况。

内部类 **LogEntry** 和 **LogFileHandler** 都与实现扫描器日志文件相关。由于数据块的扫描周期很长（默认是 21 天），所以数据块扫描器需要将扫描结果保存在日志文件中（块池存储目录下的 **dnep_block_verification.log**），以防止数据节点重启后丢失扫描信息。当数据块扫描器启动时，会加载日志文件中保存的扫描结果信息。

在上一节中我们介绍了 **DataBlockScanner** 会周期性地调用 **BlockPoolSliceScanner.scanBlockPoolSlice()** 方法验证这个块池下的所有数据块，**scanBlockPoolSlice()** 方法会调用 **scan()** 方法执行真正的扫描操作，然后修改 **DataBlockScanner.lastScanTime** 字段更新当前块池的扫描时间。**scan()** 方法会从 **DataBlockScanner.blockInfoSet** 对象中取出一个数据块，然后调用 **verifyBlock()** 方法验证这个数据块的正确性，最后将验证完成的数据块加入 **DataBlockScanner.processedBlocks** 集合中保存。数据块扫描的执行流程请参考图 4-59。

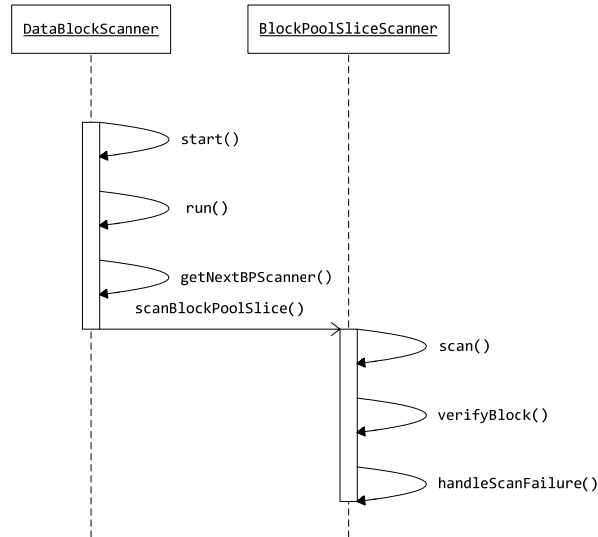


图 4-59 scanBlockPoolSlice()调用流程图

verifyBlock()方法验证数据块的逻辑也很简单，它构造一个 BlockSender 对象，然后通过 BlockSender 对象将数据块发送到一个空的数据流中。BlockSender 在发送数据块数据时，会对发送的数据进行校验，如果校验失败则抛出异常。verifyBlock()在调用 BlockSender.sendBlock()方法时，会监听 sendBlock()方法是否抛出异常。如果抛出异常，则证明这个数据块的校验和出现错误；如果两次检查数据块都出现错误，verifyBlock()则会调用 handleScanFailure()通知 Namenode 这个数据块已经损坏；如果没有抛出异常，则证明数据块是正常的，这时 verifyBlock()会调用 updateScanStatus()更新 blockInfoSet 集中数据块对应的 BlockScanInfo 对象，然后在扫描日志中记录。verifyBlock()方法的代码如下：

```

void verifyBlock(ExtendedBlock block) {
    BlockSender blockSender = null;
    for (int i=0; i<2; i++) {
        boolean second = (i > 0);
        try {
            adjustThrottler(); // 调整节流器，控制验证数据块占用 IO 的带宽

            // 构造 BlockSender 对象，将数据块发送到空的输出流中
            blockSender = new BlockSender(block, 0, -1, false, true, true,
                datanode, null, CachingStrategy.newDropBehind());
            DataOutputStream out =
                new DataOutputStream(new IOUtils.NullOutputStream()); // 空数据流
            blockSender.sendBlock(out, null, throttler);

            // 数据块验证正确，调用 updateScanStatus() 方法更新数据块扫描状态
            updateScanStatus((BlockScanInfo)block.getLocalBlock(),
                ScanType.VERIFICATION_SCAN, true);
        }
    }
}

```

```
        return;
    } catch (IOException e) {
        updateScanStatus((BlockScanInfo)block.getLocalBlock(),
            ScanType.VERIFICATION_SCAN, false);

        // 如果 FsDataset 上已经不存在这个数据块的引用, 则从磁盘上删除这个数据块
        if (!dataset.contains(block)) {
            deleteBlock(block.getLocalBlock());
            return;
        }

        if (e instanceof FileNotFoundException) {
            deleteBlock(block.getLocalBlock());
            return;
        }

        // 如果两次检查数据块的校验和都出错了
        // 则调用 handleScanFailure() 方法通知 Namenode 这个数据块已经损坏
        if (second) {
            totalScanErrors++;
            handleScanFailure(block);
            return;
        }
    } finally {
        IOUtils.closeStream(blockSender);
        datanode.getMetrics().incrBlocksVerified();
        totalScans++;
    }
}
}
```

handleScanFailure()方法的实现也很简单, 最终通过 RPC 接口 DatanodeProtocol.reportBadBlocks()向 Namenode 汇报了损坏的数据块。

4.7 DirectoryScanner

DirectoryScanner 的主要任务是定期扫描磁盘上的数据块, 检查磁盘上的数据块信息是否与 FsDatasetImpl 中保存的数据块信息一致, 如果不一致则对 FsDatasetImpl 中的信息进行更新。注意, DirectoryScanner 只会检查内存和磁盘上 FINALIZED 状态的数据块是否一致。

我们首先看一下 DirectoryScanner 定义了哪些重要的字段, 如图 4-60 所示。

- reportCompileThreadPool: 异步收集磁盘上数据块信息的线程池。
- masterThread: 主线程, 定期调用 DirectoryScanner.run()方法, 执行整个扫描逻辑。
- diffs: 描述磁盘上保存的数据块信息与内存之间差异的数据结构, 在扫描的过程中更新, 扫描结束后把 diffs 更新到 FsDatasetImpl 对象上。

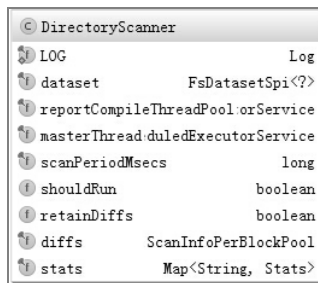


图 4-60 DirectoryScanner 定义的字段

DirectoryScanner 对象会定期（由 `dfs.datanode.directoryscan.interval` 配置，默认为 21600 秒）在线程池对象 `masterThread` 上触发扫描任务，这个扫描任务是由 `DirectoryScanner.reconcile()` 方法执行的。`reconcile()` 会首先调用 `scan()` 方法收集磁盘上数据块与内存中数据块的差异信息，并把这些差异信息保存在 `diffs` 字段中。`scan()` 方法在获取磁盘上存储的数据块时使用了 `reportCompileThreadPool` 线程池，异步地完成磁盘数据块的扫描任务。`reconcile()` 方法拿到 `scan()` 更新的 `diffs` 对象后，调用 `FsDataset` 的 `checkAndUpdate()` 方法（请参考 `FsDatasetImpl` 实现小节），更新 `FsDatasetImpl` 保存的数据块副本信息，完成与磁盘上数据块副本的同步操作。`reconcile()` 方法的代码如下：

```
void reconcile() throws IOException {
    scan(); // 调用 scan() 方法收集磁盘上数据块与内存中数据块的差异信息
    // 遍历差异信息 diff，然后处理这个差异
    for (Entry<String, LinkedList<ScanInfo>> entry : diffs.entrySet()) {
        String bpid = entry.getKey();
        LinkedList<ScanInfo> diff = entry.getValue();

        // 调用 FsDatasetImpl.checkAndUpdate() 更新 FsDataset 保存的数据块，完成同步
        for (ScanInfo info : diff) {
            dataset.checkAndUpdate(bpid, info.getBlockId(), info.getBlockFile(),
                info.getMetaFile(), info.getVolume());
        }
    }
    if (!retainDiffs) clear();
}
```

4.8 DataNode 类的实现

`DataNode` 类封装了整个数据节点逻辑的实现。它通过 `DataStorage` 以及 `FsDatasetImpl` 管理着数据节点存储上的所有数据块，`DataNode` 类还会通过流式接口对客户端和其他数据节点提供读数据块、写数据块、复制数据块等功能。同时 `DataNode` 类实现了 `InterDatanodeProtocol` 以及 `ClientDatanodeProtocol`，使得数据节点可以接收来自其他数据节点以及客户端的远程 RPC 请求。`DataNode` 类还会通过 `BlockPoolManager` 对象周期性地向 `Namenode` 发送心跳、块

汇报、增量块汇报以及缓存汇报，同时执行 Namenode 发回的名字节点指令。DataNode 持有 DataBlockScanner 对象周期性地检查存储上的所有数据块，以及 DirectoryScanner 对象验证存储上数据块和内存中数据块的一致性。介绍完了 DataNode 的所有支撑类，我们在本节介绍 DataNode 类的实现。

4.8.1 DataNode 的启动

DataNode 启动流程的入口方法是 main()方法，这个方法调用了 secureMain()方法，secureMain()方法通过调用 createDataNode()创建并启动一个 DataNode 实例，然后在这个 DataNode 实例上调用 join()方法等待 DataNode 停止运行。

createDataNode()方法首先调用静态方法 instantiateDataNode()创建 DataNode 实例，然后调用 runDatanodeDaemon()方法启动 DataNode 上的各个服务。我们依次看一下 instantiateDataNode()方法以及 runDatanodeDaemon()方法的实现。

静态方法 instantiateDataNode()首先解析 DataNode 的启动参数，获取 DataNode 配置文件中定义的所有存储目录，然后调用静态方法 makeInstance()。makeInstance()方法确保定义的存储目录至少有一个可用，然后调用 DataNode 的构造方法创建 DataNode 实例。整个 DataNode 初始化的调用顺序请参考图 4-61。

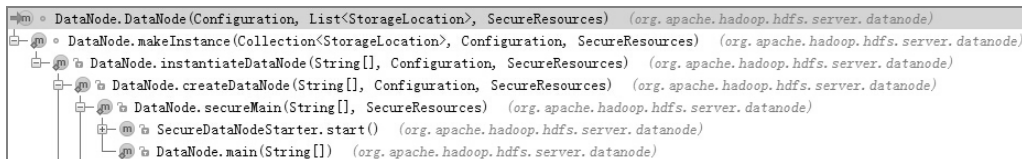


图 4-61 DataNode 初始化调用顺序

DataNode 的构造函数在初始化了若干配置文件中定义的参数后，调用 startDataNode()方法完成 DataNode 的初始化操作，startDataNode()方法初始化了 DataStorage 对象、DataXceiverServer 对象、shortCircuitRegistry 对象，启动了 HttpInfoServer，初始化了 DataNode 的 IPC Server，然后创建 BlockPoolManager 并加载每个块池定义的 Namenode 列表。startDataNode()方法的代码如下：

```
void startDataNode(Configuration conf,
                  List<StorageLocation> dataDirs,
                  SecureResources resources
                  ) throws IOException {

    // 设置 Datanode 的存储目录
    synchronized (this) {
        this.dataDirs = dataDirs;
    }
    // 加载 Datanode 配置
    this.conf = conf;
```

```

this.dnConf = new DNConf(conf);
// ...

storage = new DataStorage(); // 构造 DataStorage 对象
registerMXBean();
initDataXceiver(conf);        // 创建 DataXceiverServer 对象
startInfoServer(conf);        // 启动 HttpInfoServer 服务
pauseMonitor = new JvmPauseMonitor(conf);
pauseMonitor.start();

initIpcServer(conf); // 初始化 DataNode IPC Server

// 创建 BlockPoolManager 并加载每个块池定义的 Namenode 列表
blockPoolManager = new BlockPoolManager(this);
blockPoolManager.refreshNamenodes(conf);

// 创建 ReadaheadPool 对象
readaheadPool = ReadaheadPool.getInstance();
// ...
}

```

这里我们注意到，在 `startDataNode()` 方法中并没有初始化 `FsDatasetImpl` 对象、`DataBlockScanner` 对象以及 `DirectoryScanner` 对象，这几个对象的初始化是在 `initBlockPool()` 方法中实现的。如图 4-62 所示，`initBlockPool()` 方法是在 `BPSERVICEActor` 对象与指定命名空间中的 `Namenode` 完成握手操作之后调用的。

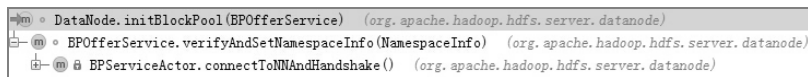


图 4-62 `initBlockPool()` 调用关系

成功初始化 `DataNode` 对象之后，就需要调用 `runDatanodeDaemon()` 方法启动 `DataNode` 的服务了。`runDatanodeDaemon()` 方法启动了 `blockPoolManager` 管理的所有线程，启动了 `DataXceiverServer` 线程，最后启动了 `DataNode` 的 `IPC Server`。需要特别注意的是，`DataBlockScanner` 线程以及 `DirectoryScanner` 线程并不是在 `runDatanodeDaemon()` 方法中启动的，而是在 `initBlockPool()` 方法中调用 `initPeriodicScanners()` 方法启动的。`runDatanodeDaemon()` 方法的代码如下：

```

public void runDatanodeDaemon() throws IOException {
    blockPoolManager.startAll();
    dataXceiverServer.start();
    if (localDataXceiverServer != null) {
        localDataXceiverServer.start();
    }
    ipcServer.start();
    startPlugins(conf);
}

```

4.8.2 DataNode 的关闭

`shutdown()`方法用于关闭 `DataNode` 实例的运行，这个方法首先将 `DataNode.shouldRun` 字段设置为 `false`，这样所有的 `BPSERVICEActor` 线程、`DataXceiverServer` 线程、`PacketResponder` 以及 `DataBlockScanner` 线程的循环条件就会不成立，线程也就自动退出运行了。如果这个关闭操作是用于重启，且当前 `Datanode` 正处于写数据流管道中，则向上游数据节点发送 `OOB` 消息通知客户端，之后调用 `DataXceiverServer.kill()`方法强行关闭流式接口底层的套接字连接。接下来 `shutdown()`方法会关闭 `DataBlockScanner` 以及 `DirectoryScanner`，关闭 `WebServer`，然后在 `DataXceiverServer` 对象上调用 `join()`方法，等待 `DataXceiverServer` 线程成功退出。最后依次关闭 `DataNode` 的 `IPC Server`、`BlockPoolManager` 对象、`DataStorage` 对象以及 `FSDatasetImpl` 对象。由于篇幅原因，这里就不贴出 `shutdown()`方法的代码了，请读者自行参考源码。

`shutdown()`方法结束运行后，数据节点上的所有服务线程也都退出了，`secureMain()`方法的 `join()`调用返回，然后 `secureMain()`方法会执行它的 `finally` 语句，并在日志系统中打印“Exiting Datanode”信息后，结束数据节点的运行并退出。`secureMain()`方法的代码如下：

```
public static void secureMain(String args[], SecureResources resources) {
    int errorCode = 0;
    try {
        // ...
        if (datanode != null) {
            datanode.join();
        } else {
            errorCode = 1;
        }
    } catch (Throwable e) {
        terminate(1, e);
    } finally {
        LOG.warn("Exiting Datanode");
        terminate(errorCode);
    }
}
```

第 5 章 HDFS 客户端

前面两章介绍了 HDFS 数据节点 (Datanode)、名字节点 (Namenode) 的实现细节,本章开始介绍 HDFS 客户端。HDFS 目前提供了三个客户端接口: DistributedFileSystem、FsShell 和 DFSAdmin。DistributedFileSystem 为用户开发基于 HDFS 的应用程序提供了 API; FsShell 工具使用户可以通过 HDFS Shell 命令执行常见的文件系统操作,例如创建文件、删除文件、创建目录等; DFSAdmin 则向系统管理员提供了管理 HDFS 的工具,例如执行升级、管理安全模式等操作。

DistributedFileSystem、FsShell 以及 DFSAdmin 都是通过直接或者间接地持有 DFSClient 对象的引用,然后调用 DFSClient 提供的接口方法对 HDFS 进行管理和操作的。DFSClient 类封装了 HDFS 复杂的交互逻辑,对外提供了简单的接口,所以本章以 DFSClient 类作为入口来研究和学习 HDFS 客户端的逻辑以及源码实现。

5.1 DFSClient 实现

DFSClient 是一个真正实现了分布式文件系统客户端功能的类,是用户使用 HDFS 各项功能的起点。DFSClient 会连接到 HDFS,对外提供管理文件/目录、读写文件以及管理与配置 HDFS 系统等功能。

对于管理文件/目录以及管理与配置 HDFS 系统这两个功能,DFSClient 并不需要与 Datanode 交互,而是直接通过远程接口 ClientProtocol 调用 Namenode 提供的服务即可。而对于文件读写功能,DFSClient 除了需要调用 ClientProtocol 与 Namenode 交互外,还需要通过流式接口 DataTransferProtocol 与 Datanode 交互传输数据。

DFSClient 对外提供的接口方法可以分为如下几类,我们在下面的小节中会依次介绍这几类方法的实现。

- DFSClient 的构造方法和关闭方法。
- 管理与配置文件系统相关方法。
- 操作 HDFS 文件与目录方法。
- 读写 HDFS 文件方法。

5.1.1 构造方法

如图 5-1 所示, DFSClient 提供了 5 个重载的构造方法, 其中参数为 Configuration 的构造方法已经弃用, 其他 4 个构造方法最终都调用了最后一个参数最多的构造方法。

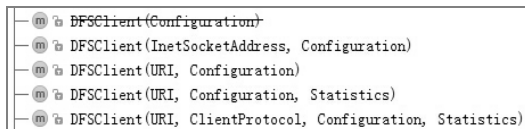


图 5-1 DFSClient 构造方法

DFSClient 类的构造方法主要完成两个功能。

- 读入配置文件, 并初始化以下成员变量。
 - conf: HDFS 配置信息。
 - stats: Client 状态统计信息, 包括 Client 读、写字节数等。
 - dtpReplaceDatanodeOnFailure: 当 Client 读写数据时, 如果 Datanode 出现故障, 是否进行 Datanode 替换的策略。
 - localInterfaceAddr: 本地接口地址。
 - readahead: 预读取字节数。
 - readDropBehind: 读取数据后, 是否立即从操作系统缓冲区中删除。
 - writeDropBehind: 写数据后, 是否立即从操作系统缓冲区中删除。
 - hedgedReadThresholdMillis: hedgedReadThresholdMillis 字段保存了触发“hedged read”机制的时长。“hedged read”模式是 Hadoop 2.4 中引入的新特性, 当 Client 发现一个数据块读取操作太慢时 (读取时长超过 hedgedReadThresholdMillis), 那么 Client 会启动另一个并发操作读取数据块的另一个副本, 之后 Client 会返回先完成读取副本的数据。
- 获取 Namenode 的 RPCProxy 引用(请参考第 2 章), 供 DFSClient 远程调用 Namenode RPC 方法。

有了上面的分析, 我们看一下 DFSClient 构造方法的代码实现。

```

public DFSClient(URI nameNodeUri, ClientProtocol rpcNamenode, Configuration conf,
FileSystem.Statistics stats) throws IOException {
    // 初始化属性
    this.dfsClientConf = new Conf(conf);
    this.conf = conf;
    this.stats = stats;
    this.socketFactory = NetUtils.getSocketFactory(conf, ClientProtocol.class);
    this.dtpReplaceDatanodeOnFailure = ReplaceDatanodeOnFailure.get(conf);

    // 获取 Namenode RPC Proxy 引用
    if (proxyInfo != null) {
        // ...
    }
}
  
```

```

} else if (rpcNamenode != null) {
    // ...
} else {
    // 通过 NameNodeProxies.createProxy() 创建 Namenode RPC 引用
    proxyInfo = NameNodeProxies.createProxy(conf, nameNodeUri,
        ClientProtocol.class);
    // 对属性赋值
    this.dtService = proxyInfo.getDelegationTokenService();
    this.namenode = proxyInfo.getProxy();
}

String localInterfaces[] =
    conf.getTrimmedStrings(DFSConfigKeys.DFS_CLIENT_LOCAL_INTERFACES);
localInterfaceAddr = getLocalInterfaceAddr(localInterfaces);

// 初始化 caching 相关属性
Boolean readDropBehind = (conf.get(DFS_CLIENT_CACHE_DROP_BEHIND_READS) == null) ?
    null : conf.getBoolean(DFS_CLIENT_CACHE_DROP_BEHIND_READS, false);
Long readahead = (conf.get(DFS_CLIENT_CACHE_READAHEAD) == null) ?
    null : conf.getLong(DFS_CLIENT_CACHE_READAHEAD, 0);
Boolean writeDropBehind = (conf.get(DFS_CLIENT_CACHE_DROP_BEHIND_WRITES) == null) ?
    null : conf.getBoolean(DFS_CLIENT_CACHE_DROP_BEHIND_WRITES, false);
this.defaultReadCachingStrategy =
    new CachingStrategy(readDropBehind, readahead);
this.defaultWriteCachingStrategy =
    new CachingStrategy(writeDropBehind, readahead);
this.clientContext = ClientContext.get(
    conf.get(DFS_CLIENT_CONTEXT, DFS_CLIENT_CONTEXT_DEFAULT),
    dfsClientConf);

// 初始化 hedgedRead 相关属性
this.hedgedReadThresholdMillis = conf.getLong(
    DFSConfigKeys.DFS_DFSCIENT_HEDGED_READ_THRESHOLD_MILLIS,
    DFSConfigKeys.DEFAULT_DFSCIENT_HEDGED_READ_THRESHOLD_MILLIS);
int numThreads = conf.getInt(
    DFSConfigKeys.DFS_DFSCIENT_HEDGED_READ_THREADPOOL_SIZE,
    DFSConfigKeys.DEFAULT_DFSCIENT_HEDGED_READ_THREADPOOL_SIZE);
if (numThreads > 0) {
    this.initThreadsNumForHedgedReads(numThreads);
}
// ...
}

```

5.1.2 关闭方法

DFSClient 的 close() 方法比较简单，它首先调用 closeAllFilesBeingWritten() 关闭所有正在进行写操作的 IO 流。接下来将 clientRunning 标志位置为 false，停止 DFSClient 对外服务，然

后停止租约管理器，最后关闭与 Namenode 的 RPC 连接。

这里 `clientRunning` 标志位标识了 `DFSCClient` 的运行状态，`clientRunning` 标志位为 `true` 时，客户端处于运行状态。在后面的小节中我们会介绍，只有 `clientRunning` 标志位为 `true` 时，`DFSCClient` 对象打开的输入流、输出流才可以正常工作。

`close()` 方法的代码实现也比较简单，如下所示。

```
public synchronized void close() throws IOException {
    if (clientRunning) {
        closeAllFilesBeingWritten(false);
        clientRunning = false;
        getLeaseRenewer().closeClient(this);
        // 关闭与 Namenode 之间的 RPC 连接
        closeConnectionToNamenode();
    }
}
```

5.1.3 文件系统管理与配置方法

HDFS 管理员通过 `DFSAdmin` 工具管理与配置 HDFS，`DFSAdmin` 也是通过持有 `DistributedFileSystem` 对象的引用，然后进一步调用 `DFSCClient` 类提供的方法执行管理与配置操作的。整个流程比较简单，这里我们以 `DFSCClient.rollEdits()` 方法为例，`DFSCClient.rollEdits()` 方法的调用序列如图 5-2 所示。

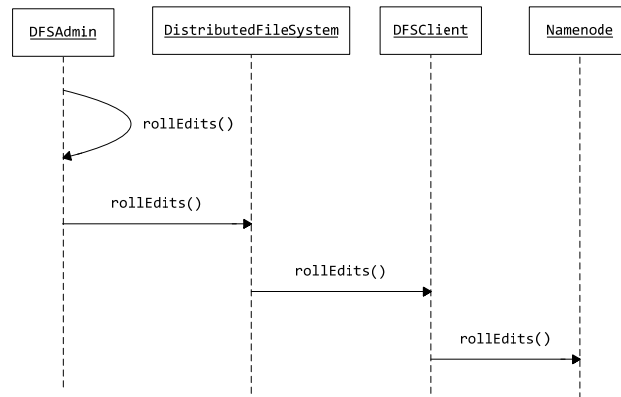


图 5-2 `rollEdits()` 调用序列图

我们看一下 `DFSCClient.rollEdits()` 方法的实现。`rollEdits()` 方法调用了 Namenode 与 Client 之间的 RPC 接口——`ClientProtocol.rollEdits()` 方法。

```
long rollEdits() throws AccessControlException, IOException {
    try {
        // 通过 ClientProtocol 接口触发 Namenode 进行 rollEdits 操作
        return namenode.rollEdits();
    }
}
```



```
    } catch (RemoteException re) {  
        throw re.unwrapRemoteException(AccessControlException.class);  
    }  
}
```

DFSAdmin 类中管理与配置文件系统的方法流程大多与 rollEdits()类似，它们直接调用了 DFSCliient 对应的方法，再由 DFSCliient 调用 ClientProtocol 对应的方法。表 5-1 给出了 DFSAdmin 参数、DFSCliient 对应的处理方法、底层支持的 ClientProtocol 接口方法的对应关系。

表 5-1 DFSCliient 文件系统管理与配置方法对应关系

DFSAdmin 参数	DFSCliient	ClientProtocol	作用
-report	datanodeReport()	getDatanodeReport()	获取当前集群中所有 Datanode 的信息
-safemode	setSafeMode()	setSafeMode()	进入、离开或者查询安全模式。处于安全模式中的 HDFS，命名空间是只读的
-allowSnapshot	allowSnapshot()	allowSnapshot()	开启指定目录的快照（snapshot）功能
-disallowSnapshot	disallowSnapshot()	disallowSnapshot()	关闭指定目录的快照功能
-saveNamespace	saveNamespace()	saveNamespace()	保存当前 Namenode 元数据到 fsimage
-rollEdits	rollEdits()	rollEdits()	提交当前的 edits_inprogress 文件
-restoreFailedStorage	restoreFailedStorage()	restoreFailedStorage()	开启或者关闭失败存储的恢复
-refreshNodes	refreshNodes()	refreshNodes()	刷新 hosts 以及 exclude 文件
-finalizeUpgrade	finalizeUpgrade()	finalizeUpgrade()	提交升级
-rollingUpgrade	rollingUpgrade()	rollingUpgrade()	进行升级
-clrQuota	setQuota()	setQuota	重置命名空间（Namespace）配额，和 setQuota()使用同样的 ClientProtocol 接口
-setQuota	setQuota()	setQuota()	设置命名空间配额
-clrSpaceQuota	setQuota()	setQuota()	重置磁盘空间配额，和 setSpaceQuota()使用同样的 ClientProtocol 接口
-setSpaceQuota	setQuota()	setQuota()	设置磁盘空间配额
-setBalancerBandwidth	setBalancerBandwidth()	setBalancerBandwidth()	设置平衡操作时的带宽

DFSCliient 中还有许多命令是直接建立与 Datanode 或者 Namenode 的 RPC 连接，然后调用对应的 RPC 方法实现的，例如 getDatanodeInfo()、shutdownDatanode()、deleteBlockPool()、refreshNamenodes()等操作。这里就不再叙述了，感兴趣的读者可以自行查阅代码。

5.1.4 HDFS 文件与目录操作方法

除了管理与配置 HDFS 文件系统外，DFSCliient 的另一个重要功能就是操作 HDFS 文件与目录，例如 setPermission()、rename()、getFileInfo()、delete()等对文件/目录树的增、删、改、查等操作。

这些方法的实现也比较简单，它们都是首先调用 checkOpen()检查 DFSCliient 的运行情况，然后调用 ClientProtocol 对应的 RPC 方法，触发 Namenode 更改文件系统目录树。

这里我们以 `DFSClient.rename()` 方法为例。`rename()` 方法调用 `ClientProtocol.rename2()` 方法向 `Namenode` 发起重命名请求, `Namenode` 接收到重命名请求后, 会直接更改命名空间中对应文件的文件名, 完成重命名操作。`DFSClient.rename()` 方法的代码如下:

```
public void rename(String src, String dst, Options.Rename... options)
    throws IOException {
    // 检查 DFSClient 的运行情况
    checkOpen();
    try {
        // 调用 ClientProtocol.rename2() 方法发起重命名请求
        namenode.rename2(src, dst, options);
    } catch (RemoteException re) {
        throw re.unwrapRemoteException(AccessControlException.class,
                                         DSQuotaExceededException.class,
                                         FileAlreadyExistsException.class,
                                         FileNotFoundException.class,
                                         ParentNotDirectoryException.class,
                                         SafeModeException.class,
                                         NSQuotaExceededException.class,
                                         UnresolvedPathException.class,
                                         SnapshotAccessControlException.class);
    }
}
```

5.1.5 HDFS 文件读写方法

`DFSClient` 中剩余的方法主要是对文件读写操作的支持, 这里涉及的方法比较多, 我们在后面内容中会重点介绍文件的读写操作, 以及读写操作对应的输入流与输出流, 本节就不再重复介绍了。

5.2 文件读操作与输入流

在 `HDFS` 客户端实现中, 最重要也最复杂的一部分就是文件的读写操作。文件读操作相对于写操作要简单一些, 本节介绍 `HDFS` 客户端文件读操作的代码实现。

5.2.1 打开文件

当用户读取一个 `HDFS` 文件时, 首先会调用 `DistributedFileSystem.open()` 方法打开这个文件, 并获取文件对应的 `FSDDataInputStream` 输入流, 然后在这个 `FSDDataInputStream` 对象上调用 `read()` 方法读取数据。本节就以 `DistributedFileSystem.open()` 方法作为代码分析的入口, 学习打开 `HDFS` 文件并获取输入流的过程。

1. DistributedFileSystem.open()

DistributedFileSystem.open()方法用于打开一个 HDFS 文件，并返回这个文件对应的 FSDataInputStream 输入流。open()首先会调用 DFSClient.open()方法创建 HDFS 文件对应的 DFSInputStream 输入流对象，然后构造一个 HdfsDataInputStream 对象包装 DFSClient 返回的 DFSInputStream，最后将这个 HdfsDataInputStream 对象返回给客户端代码。open()方法的流程如图 5-3 所示。

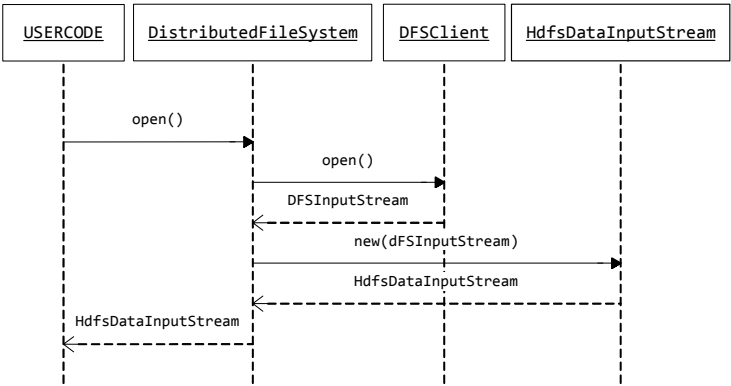


图 5-3 open()方法流程图

2. HDFS 客户端输入流

在上面介绍的 DistributedFileSystem.open()方法实现中涉及了多个输入流类，本节我们介绍 HDFS 客户端输入流。图 5-4 给出了 HDFS 客户端输入流的继承关系。

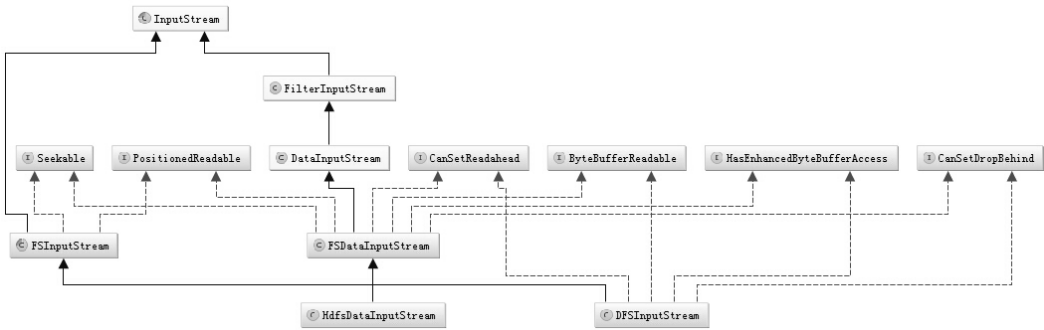


图 5-4 HDFS 客户端输入流的继承关系图

首先我们看最左侧的抽象类 FSInputStream，这个类扩展自 java.io.InputStream 类，实现了 Seekable 和 PositionedReadable 两个接口，也就是支持基于位置的读取功能。需要注意的是，FSInputStream 实现的 seek()以及 getPos()方法是抽象方法。

然后我们看 `FSInputStream` 的子类 `DFSInputStream`，它实现了 `FSInputStream` 的抽象方法，支持基于位置的读取功能。同时 `DFSInputStream` 还实现了 `ByteBufferReadable`、`CanSetDropBehind`、`CanSetReadahead`、`HasEnhancedByteBufferAccess` 四个接口，支持使用 `ByteBuffer` 接收数据、读取后删除缓存数据、预读取数据以及零拷贝读取 4 个功能。

下面我们看一下 `FSDDataInputStream` 类，这个类是 Hadoop 定义的抽象文件系统的输入流。它继承自 `java.io.DataInputStream`，底层装饰了一个 `FSInputStream` 输入流，提供了读取基本 Java 数据类型的功能。这个类实现了 `Seekable`、`PositionedReadable`、`ByteBufferReadable`、`HasFileDescriptor`、`CanSetDropBehind`、`CanSetReadahead`、`HasEnhancedByteBufferAccess` 等接口，所有接口方法的实现都是调用底层包装类 `FSInputStream` 对应的方法。`FSDDataInputStream` 类的实现如下代码所示。

```
public class FSDDataInputStream extends DataInputStream
    implements Seekable, PositionedReadable,
        ByteBufferReadable, HasFileDescriptor, CanSetDropBehind, CanSetReadahead,
        HasEnhancedByteBufferAccess {

    // 构造方法对传入的 InputStream 进行判断是否实现了 Seekable 以及 PositionedReadable 接口
    public FSDDataInputStream(InputStream in) {
        super(in);
        if( !(in instanceof Seekable) || !(in instanceof PositionedReadable) ) {
            throw new IllegalArgumentException(
                "In is not an instance of Seekable or PositionedReadable");
        }
    }

    // Seekable 接口实现，支持定位操作，调用底层 inputStream 对应的方法
    // getPos()、seekToNewSource() 等方法同理
    public synchronized void seek(long desired) throws IOException {
        ((Seekable)in).seek(desired);
    }

    // PositionedReadable 接口，支持从特定位置读取数据。调用底层 inputStream 对应的方法
    // readFully() 等方法同理
    public int read(long position, byte[] buffer, int offset, int length)
        throws IOException {
        return ((PositionedReadable)in).read(position, buffer, offset, length);
    }

    // ByteBufferReadable 接口，支持将数据写入 ByteBuffer，而不只是 byte[]。调用底层 inputStream
    // 对应的方法，HasFileDescriptor、CanSetReadahead、CanSetDropBehind 等接口同理
    public int read(ByteBuffer buf) throws IOException {
        if (in instanceof ByteBufferReadable) {
            return ((ByteBufferReadable)in).read(buf);
        }
        throw new UnsupportedOperationException("Byte-buffer read unsupported by input
stream");
    }
}
```

```

    }

    // HasEnhancedByteBufferAccess 接口提供支持零拷贝的 ByteBuffer 读取功能，如果读取失败
    // 则退化为正常的读取
    public ByteBuffer read(ByteBufferPool bufferPool, int maxLength,
        EnumSet<ReadOption> opts)
        throws IOException, UnsupportedOperationException {
        try {
            return ((HasEnhancedByteBufferAccess) in).read(bufferPool,
                maxLength, opts);
        }
        catch (ClassCastException e) {
            ByteBuffer buffer = ByteBufferUtil.
                fallbackRead(this, bufferPool, maxLength);
            if (buffer != null) {
                extendedReadBuffers.put(buffer, bufferPool);
            }
            return buffer;
        }
    }

    // 模板方法，调用 HasEnhancedByteBufferAccess.read() 方法，尝试进行零拷贝读取，如果抛
    // 出异常，则退化成为正常的读取
    final public ByteBuffer read(ByteBufferPool bufferPool, int maxLength)
        throws IOException, UnsupportedOperationException {
        return read(bufferPool, maxLength, EMPTY_READ_OPTIONS_SET);
    }
}

```

DistributedFileSystem.open() 方法会返回一个 HdfsDataInputStream 对象，这个类是 FSDataInputStream 的子类，是 FSDataInputStream 在 HDFS 文件系统上的实现，它在父类的基础上添加了 getCurrentDatanode()、getCurrentBlock()、getAllBlocks()、getVisibleLength() 等方法。HdfsDataInputStream 类的实现如下代码所示。

```

public class HdfsDataInputStream extends FSDataInputStream {
    public HdfsDataInputStream(DFSInputStream in) throws IOException {
        super(in); // 调用父类 FSDataInputStream 的构造方法
    }

    // ... getCurrentBlock()、getAllBlocks()、getVisibleLength() 的代码比较简单，都是直接调用
    // DFSInputStream 对应的方法
    public DatanodeInfo getCurrentDatanode() {
        return getDFSInputStream().getCurrentDatanode();
    }
}

```

3. DFSClient.open()

上面我们介绍了用户读取一个 HDFS 文件时，需要先调用 `DistributedFileSystem.open()` 方法获取该文件的 `DFSInputStream` 输入流对象，`DistributedFileSystem.open()` 会在底层调用 `DFSClient.open()` 方法。本节主要介绍 `DFSClient.open()` 打开文件，以及构造 `DFSInputStream` 输入流的过程。

`open()` 操作调用了两个 RPC 方法，分别是 `ClientProtocol.getBlockLocations()` 方法，用于获取文件对应的所有数据块的位置信息；`ClientDatanodeProtocol.getReplicaVisibleLength()` 方法，用于获取 `Datanode` 上存储的某个数据块的真实长度。

整个 `DFSClient.open()` 方法的调用流程如图 5-5 所示。本节就按照这个调用顺序介绍每个方法的实现。

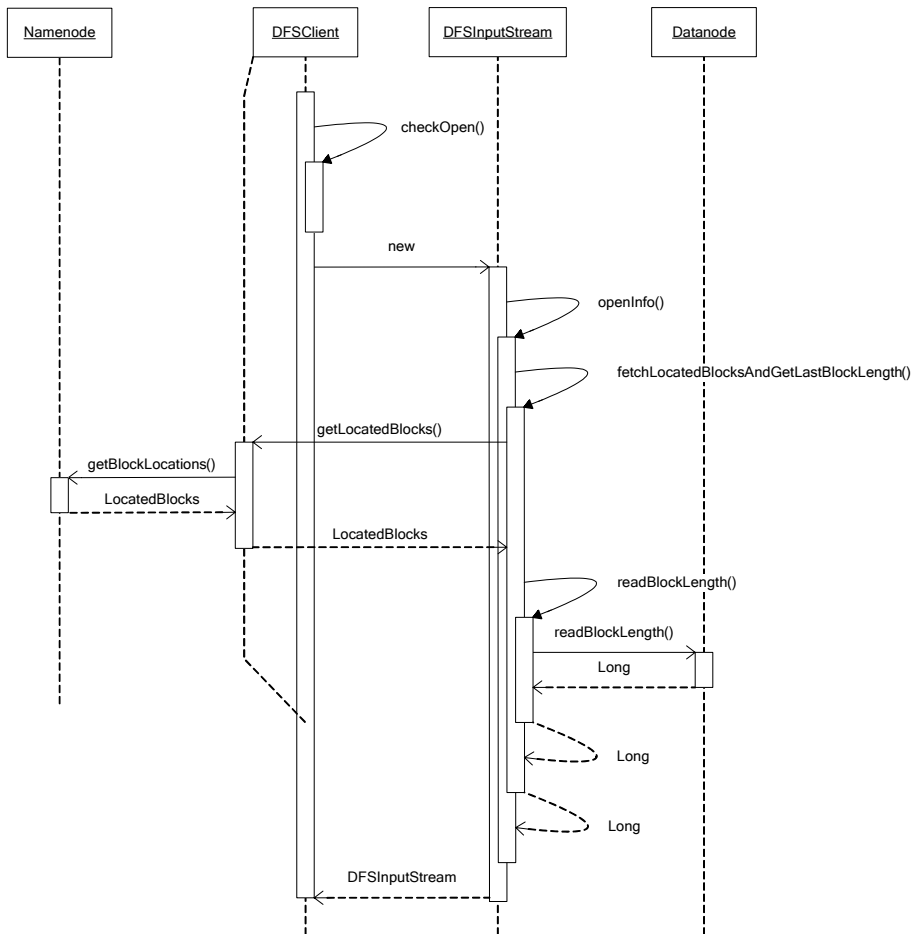


图 5-5 `open()` 方法调用流程图

(1) DFSClient.open()

DFSClient.open()用于打开文件，并获取该文件对应的输入流。这个方法的实现非常简单，首先调用 checkOpen()方法检查 DFSClient 的运行状况，然后直接调用 DFSInputStream 的构造方法并返回。

```
public DFSInputStream open(String src, int buffersize, boolean verifyChecksum)
    throws IOException, UnresolvedLinkException {
    checkOpen();
    // 调用 DFSInputStream 的构造方法并返回
    return new DFSInputStream(this, src, buffersize, verifyChecksum);
}
```

(2) DFSInputStream 构造方法

DFSInputStream 的构造方法如下代码所示，可以分为两个部分。

```
DFSInputStream(DFSClient dfsClient, String src, int buffersize, boolean verifyChecksum)
    throws IOException, UnresolvedLinkException {
    this.dfsClient = dfsClient;
    this.verifyChecksum = verifyChecksum;
    this.buffersize = buffersize;
    this.src = src;
    this.cachingStrategy =
        dfsClient.getDefaultReadCachingStrategy();
    openInfo();
}
```

■ 第一部分是初始化 DFSInputStream 的属性，包括：

- DFSClient 引用。
- verifyChecksum：读取数据时是否进行校验。
- bufferSize：读取数据时的缓冲区大小。请参考配置项 io.file.buffer.size 的配置，默认为 4KB。
- src：读取文件的地址。
- cachingStrategy：缓存策略。这里 cachingStrategy 包括两个部分——readDrop Behind，当读操作完成时，系统缓存中的数据是否立即清除，由配置项 dfs.client.cache.drop.behind.reads 配置，默认为 false；readahead，Datanode 读取时（使用了系统调用 posix_fadvise），预读取字节数，由配置项 dfs.client.cache.readahead 配置，默认为 0。

■ 第二部分是调用 openInfo()方法。

openInfo()方法用于从 Namenode 获取文件对应的数据块位置信息，并将信息保存至 DFSInputStream.locatedBlocks 字段中。openInfo()方法首先会调用 fetchLocatedBlocksAndGetLastBlockLength()方法获取文件对应的所有数据块的位置信息，然后更新当前文件的最后一个数据块的长度。由于文件的最后一个数据块可能处于构建状态（正在被写入），那么 Namenode 命名空间中保存的数据块长度就有可能小于 Datanode 实际存储数据块的长度，所以这里需要

与 Datanode 通信以确认文件最后一个数据块的真实长度。

这里还要注意一种特殊情况，当整个 HDFS 集群重启时，有些 Datanode 可能还没有向 Namenode 进行完整的数据块汇报，这时也就无法从 Namenode 获取文件对应的数据块的位置信息了。当出现这种情况时，`openInfo()` 方法将进行重试操作，重试次数由配置项 `dfs.client.retry.times.get-last-block-length` 配置，默认是重试 3 次。重试时，等待时间由 `dfs.client.retry.interval-ms.get-last-block-length` 配置，默认是 4 秒。`openInfo()` 方法的代码如下：

```
synchronized void openInfo() throws IOException, UnresolvedLinkException {
    // 获取文件对应的所有数据块的位置信息
    lastBlockBeingWrittenLength = fetchLocatedBlocksAndGetLastBlockLength();
    // 初始化重试次数
    int retriesForLastBlockLength=dfsClient.getConf().retryTimesForGetLastBlockLength;
    // 如果出现无法获取数据块长度的情况，则重试
    while (retriesForLastBlockLength > 0) {
        // 线程睡眠 4 秒，再次调用 fetchLocatedBlocksAndGetLastBlockLength() 方法
        waitFor(dfsClient.getConf().retryIntervalForGetLastBlockLength);
        lastBlockBeingWrittenLength = fetchLocatedBlocksAndGetLastBlockLength();
    } else {
        break;
    }
    retriesForLastBlockLength--;
}
if (retriesForLastBlockLength == 0) {
    // 如果重试次数过多，则直接抛出异常
    throw new IOException("Could not obtain the last block locations.");
}
}
```

如上代码所示，`openInfo()` 方法会调用 `fetchLocatedBlocksAndGetLastBlockLength()` 方法获取文件对应的所有数据块的位置信息。下面我们看一下 `fetchLocatedBlocksAndGetLastBlockLength()` 方法的实现，它的执行逻辑可以分为如下几个步骤。

- 首先调用 `dfsClient.getLocatedBlocks()` 方法通过 `ClientProtocol` 接口从 Namenode 获取当前文件对应的所有数据块的位置信息。
- 然后将新获取的位置信息与 `locatedBlocks` 字段保存的位置信息进行对比，如果出现文件数据块不匹配的情况，则抛出异常。接下来用新获取的位置信息更新 `locatedBlocks` 字段。
- 最后调用 `readBlockLength()` 方法通过 `ClientDataNodeProtocol` 接口获取文件最后一个数据块的大小，然后更新 `DFSInputStream.locatedBlocks` 字段记录的文件最后一个数据块的长度。`fetchLocatedBlocksAndGetLastBlockLength()` 方法的代码如下：

```
private long fetchLocatedBlocksAndGetLastBlockLength() throws IOException {
    // 通过 ClientProtocol 获取文件对应的所有数据块的位置信息
    final LocatedBlocks newInfo = dfsClient.getLocatedBlocks(src, 0);
    if (newInfo == null) {
        throw new IOException("Cannot open filename " + src);
    }
}
```



```

    }
    // 比较 DFSCClient.locatedBlocks 属性以及新获取的位置信息
    if (locatedBlocks != null) {
        Iterator<LocatedBlock> oldIter = locatedBlocks.getLocatedBlocks().iterator();
        Iterator<LocatedBlock> newIter = newInfo.getLocatedBlocks().iterator();
        while (oldIter.hasNext() && newIter.hasNext()) {
            // 如果数据块位置信息不匹配, 则抛出异常
            if (! oldIter.next().getBlock().equals(newIter.next().getBlock())) {
                throw new IOException("Blocklist for " + src + " has changed!");
            }
        }
    }
    // 更新 locatedBlocks 字段
    locatedBlocks = newInfo;
    long lastBlockBeingWrittenLength = 0;
    if (!locatedBlocks.isLastBlockComplete()) {
        // 获取最后一个数据块的位置信息
        final LocatedBlock last = locatedBlocks.getLastLocatedBlock();
        if (last != null) {
            if (last.getLocations().length == 0) {
                if (last.getBlockSize() == 0) {
                    //如果最后一个数据块的长度为 0, 则不用更新, 直接返回
                    return 0;
                }
                return -1;
            }
        }
        // 通过 ClientDatanodeProtocol 获取数据块在 Datanode 上的长度
        final long len = readBlockLength(last);
        // 更新 DFSCClient.locatedBlocks 保存的最后一个数据块的长度
        last.getBlock().setNumBytes(len);
        lastBlockBeingWrittenLength = len;
    }
}
// 返回结果
currentNode = null;
return lastBlockBeingWrittenLength;
}

```

在 `fetchLocatedBlocksAndGetLastBlockLength()` 方法中, 调用了 `dfsClient.getLocatedBlocks()` 以及 `readBlockLength()` 两个方法, 分别通过 `ClientProtocol` 与 `Namenode` 通信, 以及通过 `ClientDatanodeProtocol` 与 `Datanode` 通信。下面我们学习这两个方法的实现。

`dfsClient.getLocatedBlocks()` 方法直接调用了 `ClientProtocol.getBlockLocations()` 方法获取文件对应的所有数据块的位置信息。这里要注意 `prefetchSize` 参数, 它指明了这次获取数据块的范围, 由配置项 `dfs.client.read.prefetch.size` 配置, 默认是 10 个数据块大小, 如果系统数据块大小是 64MB, 那么 `prefetchSize` 大小是 640MB。`getLocatedBlocks()` 方法的返回值是文件指定范围内多个数据块的文件名以及它们的位置信息, 使用一个 `LocatedBlocks` 对象封装。对于

单个数据块的位置信息，也就是存储这个数据块副本的所有 **Datanode** 的信息，则使用 **LocatedBlock** 对象封装。

```
public LocatedBlocks getLocatedBlocks(String src, long start)
    throws IOException {
    return getLocatedBlocks(src, start, dfsClientConf.prefetchSize);
}

public LocatedBlocks getLocatedBlocks(String src, long start, long length)
    throws IOException {
    return callGetBlockLocations(namenode, src, start, length);
}

// 调用 ClientProtocol.getBlockLocations() 方法获取文件对应的所有数据块的位置信息
static LocatedBlocks callGetBlockLocations(ClientProtocol namenode,
    String src, long start, long length)
    throws IOException {
    try {
        return namenode.getBlockLocations(src, start, length);
    } catch (RemoteException re) {
        throw re.unwrapRemoteException(AccessControlException.class,
                                         FileNotFoundException.class,
                                         UnresolvedPathException.class);
    }
}
```

readBlockLength() 方法则会遍历存储这个数据块的所有 **Datanode**，然后通过调用 **ClientDatanodeProtocol.getReplicaVisibleLength()** 方法获得数据块在这个 **Datanode** 上的长度并返回。如果抛出异常，则重试。

```
private long readBlockLength(LocatedBlock locatedblock) throws IOException {
    assert locatedblock != null : "LocatedBlock cannot be null";
    int replicaNotFoundCount = locatedblock.getLocations().length;
    // 遍历保存这个数据块副本的所有数据节点
    for(DatanodeInfo datanode : locatedblock.getLocations()) {
        ClientDatanodeProtocol cdp = null;
        try {
            // 创建数据节点对应的 ClientDatanodeProtocol 对象
            cdp = DFSUtil.createClientDatanodeProtocolProxy(datanode,
                dfsClient.getConfiguration(), dfsClient.getConf().socketTimeout,
                dfsClient.getConf().connectToDnViaHostname, locatedblock);
            // 调用 ClientDatanodeProtocol.getReplicaVisibleLength() 方法获取数据块副本在当前数据
            节点上的长度
            final long n = cdp.getReplicaVisibleLength(locatedblock.getBlock());
            if (n >= 0) {
                return n;
            }
        }
        catch(IOException ioe) {
```

```

        if (ioe instanceof RemoteException &&
            ((RemoteException) ioe).unwrapRemoteException() instanceof
                ReplicaNotFoundException) {
            replicaNotFoundCount--;
        }
    } finally {
        if (cdp != null) {
            RPC.stopProxy(cdp);
        }
    }
}
if (replicaNotFoundCount == 0) {
    return 0;
}
throw new IOException("Cannot obtain block length for " + locatedblock);
}

```

至此，DFSInputStream 构造完毕，DFSClient.open()方法会将这个 DFSInputStream 对象包装成 HdfsDataInputStream 对象并返回给客户端，之后客户端就可以调用 HdfsDataInputStream.read()方法读取 HDFS 文件了。

5.2.2 读操作——DFSInputStream 实现

HDFS 目前实现的读操作有三个层次，分别是网络读、短路读（short circuit read）以及零拷贝读（zero copy read），它们的读取效率依次递增。

- 网络读：网络读是最基本的一种 HDFS 读，DFSClient 和 Datanode 通过建立 Socket 连接传输数据。
- 短路读：当 DFSClient 和保存目标数据块的 Datanode 在同一个物理节点上时，DFSClient 可以直接打开数据块副本文件读取数据，而不需要 Datanode 进程的转发。我们会在文件短路读操作小节中介绍短路读的实现。
- 零拷贝读：当 DFSClient 和缓存目标数据块的 Datanode 在同一个物理节点上时，DFSClient 可以通过零拷贝的方式读取该数据块，大大提高了效率。而且即使在读取过程中该数据块被 Datanode 从缓存中移出了，读取操作也可以退化成本地短路读，非常方便。

HdfsDataInputStream.read()方法就实现了上面描述的层次读取。HdfsDataInputStream.read()方法首先会调用 HasEnhancedByteBufferAccess.read()方法尝试进行零拷贝读取，如果当前配置不支持零拷贝读取模式，则抛出异常，然后调用 ByteBufferUtil.fallbackRead()静态方法退化成短路读或者网络读。HdfsDataInputStream.read()方法的调用流程如图 5-5 所示。

HdfsDataInputStream 实现了 HasEnhancedByteBufferAccess.read() 方法以及 InputStream.read()方法（请参考 HDFS 客户端输入流小节），这两个方法的实现都是通过调用底层包装类 DFSInputStream 对应的方法执行的。HasEnhancedByteBufferAccess.read()方法定义了零

拷贝读取的实现，而 `InputStream.read()` 方法则定义了短路读和网络读的实现。这里我们就对 `DFSInputStream` 实现的 `InputStream.read()` 方法以及 `HasEnhancedByteBufferAccess.read()` 方法进行讲解。

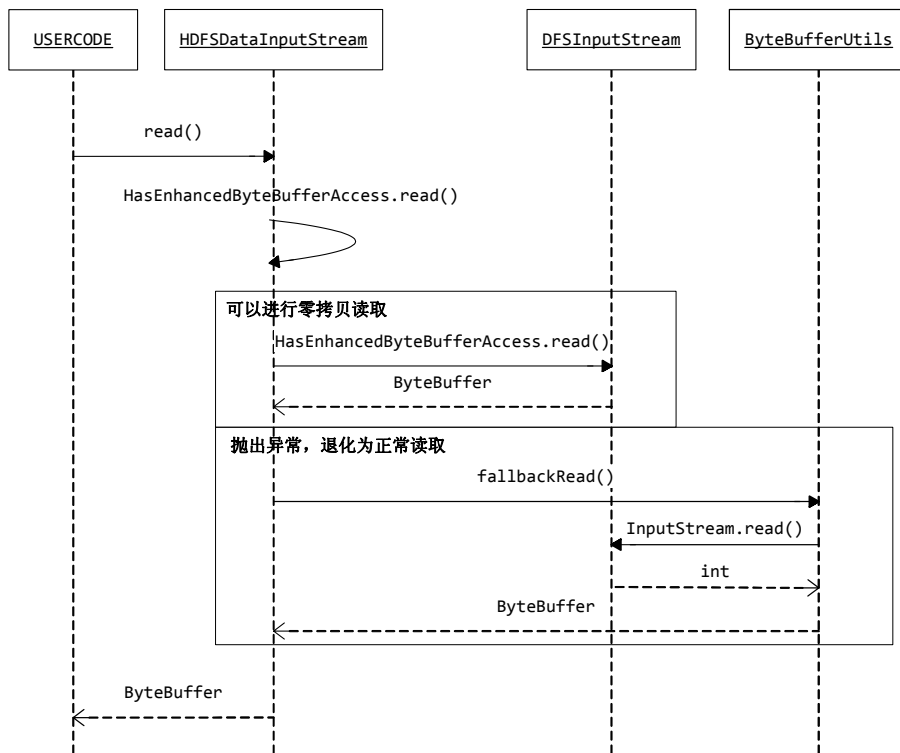


图 5-6 HdfsDataInputStream.read()方法调用流程图

1. InputStream.read()

`DFSInputStream` 实现的 `InputStream.read()` 方法的流程如图 5-7 所示。本节就按照流程图的调用顺序来介绍。

`DFSInputStream` 实现的 `InputStream.read()` 方法如下代码所示。`InputStream.read()` 方法将从输入流的 off 游标开始，读取 len 个字节，然后存入 buf[] 缓存数组中，这里的 off、len 以及 buf[] 都是 read() 方法的输入参数。read() 方法首先会构造一个 `ByteArrayStrategy` 对象，表明当前的读取操作使用字节数组作为容器，然后调用 `readWithStrategy()` 方法读取数据。

```

public synchronized int read(final byte buf[], int off, int len) throws IOException {
    // 这里使用字节数组作为容器
    ReaderStrategy byteArrayReader = new ByteArrayStrategy(buf);
    return readWithStrategy(byteArrayReader, off, len);
}
  
```

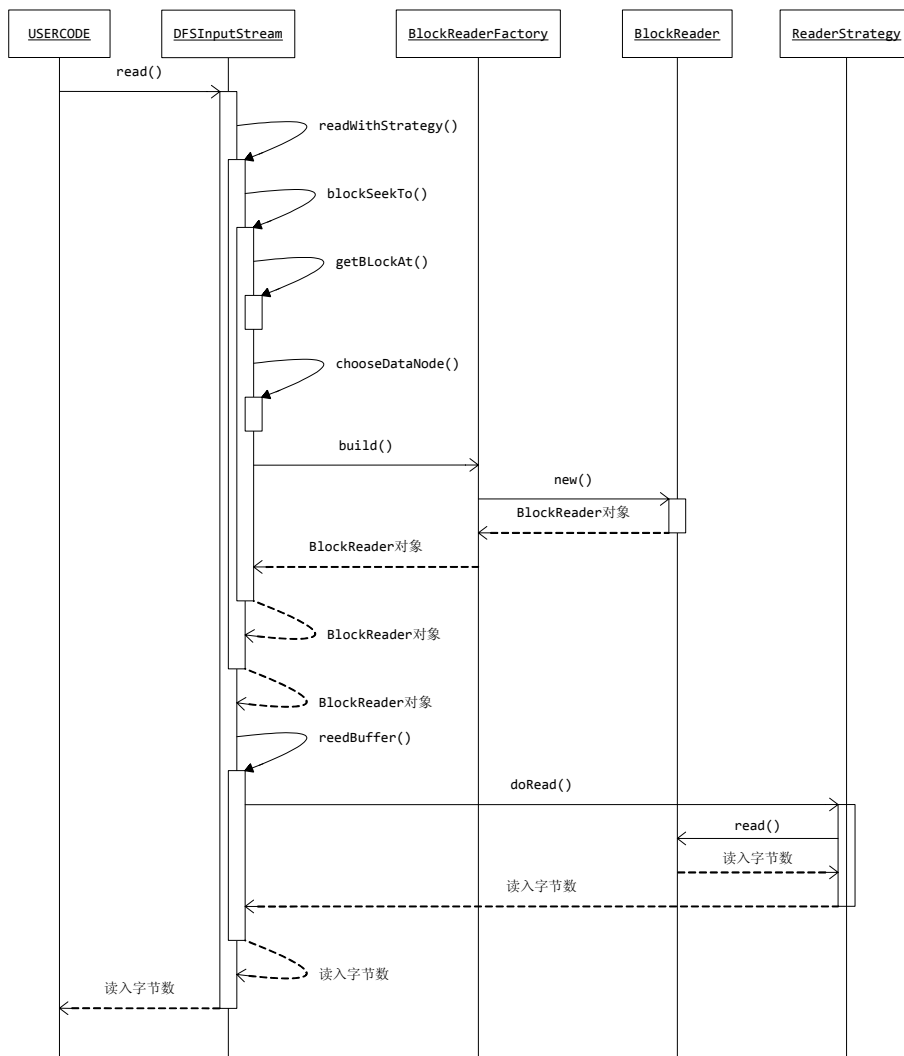


图 5-7 InputStream.read()流程图

下面我们看一下 `readWithStrategy()` 方法的实现。`readWithStrategy()` 方法首先会调用 `blockSeekTo()` 方法获取一个保存了目标数据块的 `Datanode`，然后调用 `readBuffer()` 方法从该 `Datanode` 读取数据块。如果读取过程出现 IO 异常，则进行重试操作，并将该 `Datanode` 放入黑名单中。`readWithStrategy()` 方法的代码如下：

```
private int readWithStrategy(ReaderStrategy strategy, int off, int len) throws
IOException {
    // ...这里检查 dfsClient 是否已经关闭，如果关闭了，则抛出异常
```

Hadoop 2.X HDFS 源码剖析

```
// corruptedBlockMap 用于保存损坏的数据块
Map<ExtendedBlock, Set<DatanodeInfo>> corruptedBlockMap
    = new HashMap<ExtendedBlock, Set<DatanodeInfo>>();
failures = 0;
if (pos < getFileLength()) { // 读取位置在文件范围内
    int retries = 2;          // 如果出现异常，则重试两次
    while (retries > 0) {
        try {
            // pos 超过数据块边界，需要从新的数据块开始读取数据
            if (pos > blockEnd || currentNode == null) {
                // 调用 blockSeekTo() 方法获取保存这个数据块的一个数据节点
                currentNode = blockSeekTo(pos);
            }
            // 计算这次读取的长度
            int realLen = (int) Math.min(len, (blockEnd - pos + 1L));
            if (locatedBlocks.isLastBlockComplete()) {
                realLen = (int) Math.min(realLen, locatedBlocks.getFileLength());
            }

            // 调用 readBuffer() 方法读取数据
            int result = readBuffer(strategy, off, realLen, corruptedBlockMap);

            if (result >= 0) {
                pos += result; // pos 移位
            } else {
                throw new IOException("Unexpected EOS from the reader");
            }
            ...
            return result;
        } catch (ChecksumException ce) {
            throw ce; // 出现校验错误，则抛出异常
        } catch (IOException e) {
            // ...
            blockEnd = -1;
            if (currentNode != null) { addToDeadNodes(currentNode); } // 将当前失败的节点入
黑名单中
            if (--retries == 0) { // 重试超过两次，直接抛出异常
                throw e;
            }
        } finally {
            // 检查是否需要向 Namenode 汇报损坏的数据块
            reportChecksumFailure(corruptedBlockMap,
                currentLocatedBlock.getLocations().length);
        }
    }
}
return -1;
}
```

可以看到，`readWithStrategy()`调用了 `blockSeekTo()`以及 `readBuffer()`方法，我们重点学习这两个方法的实现。

(1) `blockSeekTo()`

我们知道一个 HDFS 文件会被切分成多个数据块，这些数据块分散存储在 HDFS 集群的 `Datanode` 上。当我们读取文件时，也就是按顺序读取数据块时，如果读操作完成了对一个数据块的读取，就需要构造读取下一个数据块的输入流，这时就需要调用 `blockSeekTo()`方法获取保存下一个数据块的 `Datanode`。

`blockSeekTo()`方法首先会调用 `getBlockAt()`方法获取游标(`DFSInputStream.pos` 字段保存)所在数据块的信息，然后调用 `chooseDataNode()`方法获取一个存储了该数据块的 `Datanode`。接下来会构造从这个节点读取数据块的 `BlockReader` 对象，构造的 `BlockReader` 对象会被保存在 `DFSInputStream.blockReader` 字段中。这里要注意，构造 `BlockReader` 时使用了 `BlockReaderFactory` 这个工厂类，我们会在下一节中学习 `BlockReader` 类的实现。

注：`BlockReaderFactory` 是构造 `BlockReader` 的工厂类，这里使用了 `Builder` 模式。如果类的构造器或者静态工厂中具有多个参数，那么在设计这种类时，`Builder` 模式就是一种不错的选择，特别是当大多数参数都是可选时。与使用传统的重叠构造器模式相比，使用 `Builder` 模式的客户端代码更易于阅读和编写，构造器也比 `JavaBeans` 更加安全。

`blockSeekTo()`方法的代码如下：

```
private synchronized DatanodeInfo blockSeekTo(long target) throws IOException {
    if (target >= getFileLength()) { // 如果读取位置超过 HDFS 文件长度，则抛出异常
        throw new IOException("Attempted to read past end of file");
    }
    if (blockReader != null) { // 关闭上一个数据块对应的 BlockReader
        blockReader.close();
        blockReader = null;
    }

    // 使用 chosenNode 记录选中的 Datanode
    DatanodeInfo chosenNode = null;
    // ... 安全相关
    boolean connectFailedOnce = false;

    while (true) {
        LocatedBlock targetBlock = getBlockAt(target, true); // 获取 target 对应的数据块的位置信息
        long offsetIntoBlock = target - targetBlock.getStartOffset(); // 获取当前 target 在新数据块中的偏移量

        // 调用 chooseDataNode() 方法，获取一个 Datanode 用来读取该数据块
        DNAddrPair retval = chooseDataNode(targetBlock, null);
```

```
chosenNode = retval.info;    // 选中的 Datanode
InetSocketAddress targetAddr = retval.addr; // 选中的 Datanode 的地址

try {
    ExtendedBlock blk = targetBlock.getBlock();
    Token<BlockTokenIdentifier> accessToken = targetBlock.getBlockToken();
    // 通过 BlockReaderFactory 获取 blockReader 对象
    blockReader = new BlockReaderFactory(dfsClient.getConf()).
        setInetSocketAddress(targetAddr).
        setRemotePeerFactory(dfsClient).
        // ....
        build();
    return chosenNode;
} catch (IOException ex) {
    if (ex instanceof InvalidEncryptionKeyException && refetchEncryptionKey > 0) {
        // 安全相关的异常
    } else if (refetchToken > 0 && tokenRefetchNeeded(ex, targetAddr)) {
        // 安全相关
    } else {
        // BlockReader 构造失败, 将 chosenNode 放入黑名单中
        addToDeadNodes(chosenNode);
    }
}
}
```

下面我们就分析 `getBlockAt()` -> `chooseDataNode()` -> `blockReader`, 也就是获取数据块 -> 获取数据块对应的数据节点 -> 获取 `BlockReader` 对象的过程。

- **getBlockAt():** 该方法用于获取文件 pos 游标所在数据块的位置信息, 也就是获取该数据块对应的 `LocatedBlock` 对象。`LocatedBlock` 对象保存了所有存储该数据块的 `Datanode` 信息, 这些信息会按照距离客户端的远近排序, 同时 `LocatedBlock` 还保存了当前数据块是否被缓存等信息。`getBlockAt()` 方法会调用 `ClientProtocol.getBlockLocations()` 方法从 `Namenode` 获取 `LocatedBlock` 对象, 并将这个 `LocatedBlock` 对象保存到 `DFSInputStream.locatedBlocks` 字段中。
- **chooseDataNode():** 选择一个合适的 `Datanode` 读取数据块。这个方法的逻辑很简单, 由于 `LocatedBlock` 对象中已经包含了按照与客户端距离远近排序的 `Datanode` 列表, 所以只需遍历这个列表, 选出第一个不在 `Datanode` 黑名单 (`DFSInputStream.deadNodes` 字段保存) 中的 `Datanode` 即可。
- **BlockReaderFactory.build():** 构造从指定 `Datanode` 上读取数据块的 `BlockReader` 对象, 这里使用了 `BlockReaderFactory` 这个工厂类, `BlockReaderFactory.build()` 方法的实现请参考 `BlockReader` 小节。

(2) readBuffer()

`readWithStrategy()` 方法调用完成 `blockSeekTo()` 方法之后, 就会调用 `readBuffer()` 方法读入

数据。如下代码所示，`readBuffer()`的读入操作主要是通过委托 `BlockReader` 对象实现的，并在发生异常时进行重试。当读取出现校验异常时，表明 `currentNode` 上的数据块出现了错误，这时 `readBuffer()` 方法会将错误的 数据块 添加到 `corruptedBlockMap` 中，并通过 `reportChecksumFailure()`方法向 `Namenode` 汇报错误的数据块。如果是普通的 IO 异常，则有可能是客户端与数据节点之间的连接关闭了，那么 `readBuffer()`方法会在当前节点上调用 `seekToBlockSource()`重试。如果重试失败，则调用 `seekToNewSource()`选择新的 `Datanode`，并将当前 `Datanode` 加入黑名单中。`seekToBlockSource()`和 `seekToNewSource()`方法都调用了上一节中介绍的 `blockSeekTo()`方法，这里就不再赘述了。

```
private synchronized int readBuffer(ReaderStrategy reader, int off, int len,
    Map<ExtendedBlock, Set<DatanodeInfo>> corruptedBlockMap)
    throws IOException {
    IOException ioe;
    boolean retryCurrentNode = true;
    while (true) {
        try {
            // 调用 reader.doRead() 读取数据
            return reader.doRead(blockReader, off, len, readStatistics);
        } catch (ChecksumException ce) {
            // 出现校验异常时，表明 currentNode 上的数据块出现了错误
            ioe = ce;
            retryCurrentNode = false;
            // 将损坏的数据块加入 corruptedBlockMap 中，并向 Namenode 汇报
            addToCorruptedBlockMap(getCurrentBlock(), currentNode,
                corruptedBlockMap);
        } catch (IOException e) {
            if (!retryCurrentNode) {
                // ...
            }
            ioe = e;
        }
        boolean sourceFound = false;
        if (retryCurrentNode) {
            // 重试当前节点
            sourceFound = seekToBlockSource(pos);
        } else {
            // 当前 Datanode 重试失败，则将当前节点加入黑名单中，然后重新选择一个 Datanode 读取数据
            addToDeadNodes(currentNode);
            sourceFound = seekToNewSource(pos);
        }
        if (!sourceFound) {
            throw ioe;
        }
        retryCurrentNode = false;
    }
}
```

2. BlockReader

在 `DFSInputStream.read()` 方法中，调用了 `BlockReader` 对象的 `doRead()` 方法读取数据块。本节就介绍 `BlockReader` 类的设计与实现。

`BlockReader` 是一个接口，抽象了从指定数据节点上读取数据块的类。如图 5-8 所示，`BlockReader` 主要有三个子类。

- `BlockReaderLocal`: 进行本地短路读取（请参考短路读取相关小节）的 `BlockReader`。当客户端与 `Datanode` 在同一台物理机器上时，客户端可以直接从本地磁盘读取数据，绕过 `Datanode` 进程，从而提高了读取性能。
- `BlockReaderLocalLegacy`: 老版本的 `BlockReaderLocal`。当客户端与 `Datanode` 在同一台机器上时，客户端直接从磁盘读取数据。老版本的实现要求客户端获取 `Datanode` 数据目录的权限，这可能引入安全问题（请参考 HDFS-2246）。
- `RemoteBlockReader2`: 使用 TCP 协议从 `Datanode` 读取数据块。

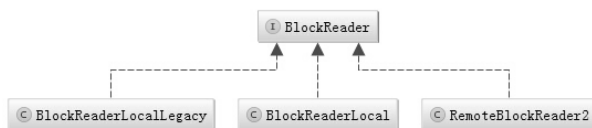


图 5-8 `BlockReader` 继承关系图

`BlockReader` 接口定义了如下方法，如图 5-9 所示。

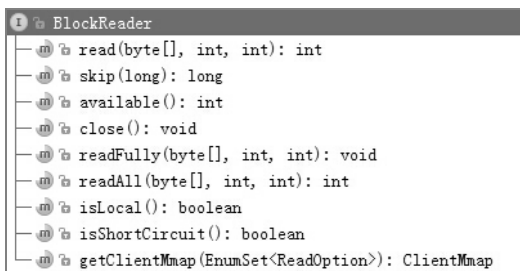


图 5-9 `BlockReader` 接口定义的方法

- `read()`、`readFully()`、`readAll()`: 将数据读取到 `byte[]` 数组中。
- `skip()`: 从数据块中跳过若干字节。
- `available()`: 当不用进行一次新的网络 IO 时，当前输入流可以读取的字节数。
- `isLocal()`: 是否是一个本地读取，也就是说，客户端和数据块是否在同一台机器上。
- `isShortCircuit()`: 是否是一个短路读取，注意短路读取必须是本地读取。
- `getClientMmap()`: 为当前读取获得一个内存映射区域（请参考零拷贝读取）。

本节首先介绍 `BlockReaderFactory` 构造 `BlockReader` 对象的流程，然后分别介绍 `RemoteBlockReader2` 和 `BlockReaderLocal` 类的实现。

(1) BlockReaderFactory

上一节中我们已经介绍了，BlockReader 的构造主要是通过 BlockReaderFactory 进行的，这一节我们就从 BlockReaderFactory.build() 构造方法开始分析 BlockReader 的构造过程。

build() 方法首先尝试创建一个本地短路读取器，短路读取避免了 Socket 通信的开销。如果短路读取方式创建失败，则创建一个域套接字读取器，这种方式使用 Linux 的 domainSocket 方法进行本地传输（由 dfs.client.domain.socket.data.traffic 配置，默认为 false）。如果上述两种方式都不能创建成功，则创建一个远程读取器，使用 TCP 进行数据的读取。

```
public BlockReader build() throws IOException {
    BlockReader reader = null;

    Preconditions.checkNotNull(configuration);
    if (conf.shortCircuitLocalReads && allowShortCircuitLocalReads) {
        if (clientContext.getUseLegacyBlockReaderLocal()) {
            reader = getLegacyBlockReaderLocal();
            if (reader != null) {
                return reader;
            }
        } else {
            reader = getBlockReaderLocal();
            if (reader != null) {
                return reader;
            }
        }
    }
    if (conf.domainSocketDataTraffic) {
        reader = getRemoteBlockReaderFromDomain();
        if (reader != null) {
            return reader;
        }
    }
    return getRemoteBlockReaderFromTcp();
}
```

getBlockReaderLocal()

getBlockReaderLocal() 方法会尝试创建一个本地短路读取器。这个方法首先从 clientContext 中获取 ShortCircuitCache，ShortCircuitCache 是在 DFSCClient 端缓存 ShortCircuitReplicaInfo 的类。然后调用 fetchOrCreate() 方法从 ShortCircuitCache 中获取当前读取数据块对应的 ShortCircuitReplicaInfo 类。

ShortCircuitCache 类我们在文件短路读操作小节中一并介绍。ShortCircuitCache 中的 ShortCircuitReplica 类保存了用来执行短路读取的文件描述符、Client 和 Datanode 共享内存中记录当前副本信息的 Slot 对象，以及数据块在内存中的映射文件 mmapData。

获取了数据块对应的 ShortCircuitReplica 后，getBlockReaderLocal() 方法会使用 ShortCircuitReplica 中保存的文件描述符构造数据块文件以及校验文件的输入流，然后构造

BlockReaderLocal 类。这里的代码比较简单，不再详细叙述，请读者自行参考相关代码。

getRemoteBlockReaderFromDomain() & getRemoteBlockReaderFromTcp()

这两个方法分别使用 Domain Socket 以及 TCP Socket 作为底层 IO 流，构造 RemoteBlockReader2 对象读取数据块。这里的代码实现也比较简单，不再赘述，请读者自行参考这两个方法的代码。

(2) RemoteBlockReader2

RemoteBlockReader2 类实现了通过 Socket 连接(可以是 Domain Socket 或者 TCP Socket)从 Datanode 读取一个数据块的逻辑。

这里要注意，在这个类的实现中，有如下几个名词。

- 数据块 (block) —— 一个 HDFS 文件数据块，往往比较大 (64MB)。
- 校验块 (chunk) —— 数据块会被切分成若干校验块，每个校验块的大小为一个校验和所验证的数据的大小。
- 数据包 (packet) —— 用来传输一组校验块的集合。一个数据包会包含一个头域，然后是所有校验块的校验和，接下来是校验块的序列。

了解清楚了这些名词之后，我们来看 RemoteBlockReader2 的具体实现。我们还是从入口方法 read() 开始。read() 方法会调用 readNextPacket() 方法读取一个数据包，并将数据包中的数据部分存入 curDataSlice 变量中，然后 read() 方法会将 curDataSlice 变量保存的数据写入 buf 中并返回。

```
public int read(ByteBuffer buf) throws IOException {
    if (curDataSlice == null || curDataSlice.remaining() == 0 && bytesNeededToFinish > 0) {
        // 读取下一个数据包，将数据包中的数据部分存入 curDataSlice 变量中
        readNextPacket();
    }
    if (curDataSlice.remaining() == 0) {
        return -1;
    }
    // 将 curDataSlice 中的数据写入 buf 中
    int nRead = Math.min(curDataSlice.remaining(), buf.remaining());
    ByteBuffer writeSlice = curDataSlice.duplicate();
    writeSlice.limit(writeSlice.position() + nRead);
    buf.put(writeSlice);
    curDataSlice.position(writeSlice.position());
    return nRead;
}
```

可以看到，read() 方法的主要实现都在 readNextPacket() 方法中，这个方法从输入流中读取一个新的数据包，将数据包中的数据部分写入 curDataSlice 变量中，并对读入的数据和数据包中的校验和部分进行匹配。我们看一下代码。

```
private void readNextPacket() throws IOException {
```

```

// 调用 packetReceiver 从 IO 流中读取一个新的数据包
packetReceiver.receiveNextPacket(in);
// 将数据包头读入 curHeader 变量中, 将数据包数据写入 curDataSlice 变量中
PacketHeader curHeader = packetReceiver.getHeader();
curDataSlice = packetReceiver.getDataSlice();
assert curDataSlice.capacity() == curHeader.getDataLen();

// 检查头域中的长度
if (!curHeader.sanityCheck(lastSeqNo)) {
    throw new IOException("BlockReader: error in packet header " +
        curHeader);
}
// 检查数据和校验和是否匹配
if (curHeader.getDataLen() > 0) {
    int chunks = 1 + (curHeader.getDataLen() - 1) / bytesPerChecksum;
    int checksumsLen = chunks * checksumSize;

    assert packetReceiver.getChecksumSlice().capacity() == checksumsLen :
        "checksum slice capacity=" + packetReceiver.getChecksumSlice().capacity() +
        " checksumsLen=" + checksumsLen;

    lastSeqNo = curHeader.getSeqno();
    if (verifyChecksum && curDataSlice.remaining() > 0) {
        checksum.verifyChunkedSums(curDataSlice,
            packetReceiver.getChecksumSlice(),
            filename, curHeader.getOffsetInBlock());
    }
    bytesNeededToFinish -= curHeader.getDataLen();
}

if (curHeader.getOffsetInBlock() < startOffset) {
    int newPos = (int) (startOffset - curHeader.getOffsetInBlock());
    curDataSlice.position(newPos);
}

// 如果完成了客户端的整个读取操作, 读取最后一个空的数据包, 因为数据块的最后一个数据包为空的标识数据包
if (bytesNeededToFinish <= 0) {
    readTrailingEmptyPacket();
    if (verifyChecksum) {
        sendReadResult(Status.CHECKSUM_OK);
    } else {
        sendReadResult(Status.SUCCESS);
    }
}
}

```

对于方法中调用的 `PacketReceiver.receiveNextPacket()` 方法, 可以参考第 4 章中对

PacketReceiver 的讲解。

(3) BlockReaderLocal

BlockReaderLocal 类实现了本地短路读取功能，也就是当客户端与 Datanode 在同一台机器上时，客户端可以绕过 Datanode 进程直接从本地磁盘读取数据。

如图 5-10 所示，当客户端向 Datanode 请求数据时，Datanode 会打开块文件以及该块文件的元数据文件，将这两个文件的文件描述符通过 domainSocket 传给客户端，客户端拿到文件描述符后构造输入流，之后通过输入流直接读取磁盘上的块文件。采用这种方式，数据绕过了 Datanode 进程的转发，提供了更好的读取性能（请参考 HDFS-347）。由于文件描述符是只读的，所以客户端不能修改收到的文件；同时由于客户端自身无法访问块文件所在的目录，所以它也就不能访问数据目录中的其他文件了，从而提高了数据读取的安全性。

UNIX Domain Socket 是一种进程间通信方式，它使得同一台机器上的两个进程能以 Socket 方式通信。它带来的另一大好处是，两个进程除了可以传递普通数据外，还可以在进程间传递文件描述符。

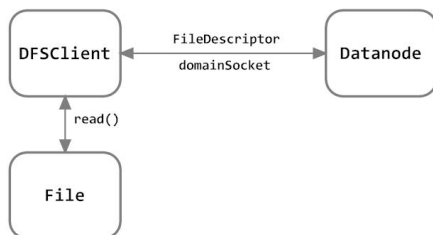


图 5-10 BlockReaderLocal 流程图

创建一个本地短路读取器有如下要求。

- 短路读相关配置项已经打开。
- DFSCClient 通过 `DataTransferProtocol.requestShortCircuitFds()` 方法获取了数据块文件以及元数据文件的文件描述符。
- DFSCClient 读取了文件描述符。

我们从入口程序 `read(ByteBuffer buf)` 方法入手，`read(byte[] arr, int off, int len)` 方法的实现与这个方法类似，只不过读取的数据保存在字节数组中。`read(ByteBuffer buf)` 方法的代码如下：

```
public synchronized int read(ByteBuffer buf) throws IOException {
    // 能否跳过数据校验
    boolean canSkipChecksum = createNoChecksumContext();
    try {
        // ...
        int nRead;
        try {
            // 可以跳过数据校验，以及不需要预读取时，调用 readWithoutBounceBuffer() 方法
            if (canSkipChecksum && zeroReadaheadRequested) {
```

```

        nRead = readWithoutBounceBuffer(buf);
    } else {
        // 需要校验, 以及开启了预读取时, 调用 readWithBounceBuffer() 方法
        nRead = readWithBounceBuffer(buf, canSkipChecksum);
    }
} catch (IOException e) {
    throw e;
}
return nRead;
} finally {
    if (canSkipChecksum) releaseNoChecksumContext();
}
}

```

`read(ByteBuffer buf)` 方法的代码可以切分为三块：① 判断能否通过 `createNoChecksumContext()` 方法创建一个免校验上下文；② 如果可以免校验，并且无预读取请求，则调用 `readWithoutBounceBuffer()` 方法读取数据；③ 如果不可以免校验，并且开启了预读取，则调用 `readWithBounceBuffer()` 方法读取数据。下面我们就看一下这三块代码的具体实现。

`createNoChecksumContext()`

`createNoChecksumContext()` 会判断如果 `verifyChecksum` 字段为 `false`，也就是当前配置本来就不需要进行校验，则直接返回 `true`，创建免校验上下文成功。如果当前配置需要进行校验，那么尝试在 `Datanode` 和 `Client` 共享内存中副本的 `Slot` 上添加一个免校验的锚（锚的概念请参考文件短路读操作小节）。这里注意，当且仅当 `Datanode` 已经缓存了这个副本时，才可以添加一个锚。因为当 `Datanode` 尝试缓存一个数据块副本时，会验证数据块的校验和，然后通过 `mmap` 以及 `mlock` 将数据块缓存到内存中。也就是说，当前 `Datanode` 上缓存的数据块是经过校验的、是正确的，不用再次进行校验。

`readWithoutBounceBuffer()`

`readWithoutBounceBuffer()` 方法非常简单，它不使用额外的数据以及校验和缓冲区预读取数据以及校验和，而是直接从数据流中将数据读取到缓冲区。

```

private synchronized int readWithoutBounceBuffer(ByteBuffer buf)
    throws IOException {
    freeDataBufIfExists(); // 释放 dataBuffer
    freeChecksumBufIfExists(); // 释放 checksumBuffer
    int total = 0;
    // 直接从输入流中将数据读取到 buf
    while (buf.hasRemaining()) {
        int nRead = dataIn.read(buf, dataPos);
        if (nRead <= 0) break;
        dataPos += nRead;
        total += nRead;
    }
    return (total == 0 && (dataPos == dataIn.size())) ? -1 : total;
}

```

readWithBounceBuffer()

`readWithBounceBuffer()`方法在 `BlockReaderLocal` 对象上申请了两个缓冲区：`dataBuf`，数据缓冲区；`checksumBuf`，校验和缓冲区。`dataBuf` 缓冲区的大小为 `maxReadaheadLength`，这个长度始终是校验块（`chunk`，一个校验值对应的数据长度）的整数倍，这样设计是为了进行校验操作时比较方便，能够以校验块为单位读取数据。`dataBuf` 和 `checksumBuf` 的构造使用了 `direct byte buffer`，也就是堆外内存上的缓冲区。

`dataBuf` 以及 `checksumBuf` 都是通过调用 `java.nio.ByteBuffer.allocateDirect()`方法分配的堆外内存。这里值得我们积累，对于比较大的缓冲区，可以通过调用 `java.nio` 提供的方法，将缓冲区分配在堆外，节省宝贵的堆内存空间。请读者参考 `org.apache.hadoop.util.DirectBufferPool` 类的实现。

`BlockReaderLocal` 提供了对缓冲区操作的几个方法。

- `fillBuffer(ByteBuffer buf, boolean canSkipChecksum)`：将数据从输入流读入指定 `buf` 中，并将校验和读入 `checksumBuf` 中进行校验操作。
- `fillDataBuf()`：调用 `fillBuffer()`方法将数据读入 `dataBuf` 缓冲区中，将校验和读入 `checksumBuf` 缓冲区中。这里要注意，`dataBuf` 缓冲区中的数据始终是 `chunk`（一个校验值对应的数据长度）的整数倍。
- `drainDataBuf(ByteBuffer buf)`：将 `dataBuf` 缓冲区中的数据拉取到 `buf` 中，然后返回读取的字节数。

了解了 `BlockReaderLocal` 定义的缓冲区方法，我们再来看 `readWithBounceBuffer()`的实现就比较简单了。首先从 `dataBuf`中拉取缓存中的数据到 `buf`，这样就保证了读取游标 `pos` 在 `chunk` 边界上。如果 `buf` 的剩余空间大于 `dataBuf` 缓冲区的大小，且当前数据流游标在 `chunk` 边界上，则调用 `fillBuffer(buf)`方法将数据直接读入 `buf`，而不通过 `dataBuf` 缓存。如果 `buf` 的剩余空间小于 `dataBuf` 缓冲区的大小，则先调用 `fillDataBuf()`方法将数据读入 `dataBuf` 缓存，然后再调用 `drainDataBuf()`将 `dataBuf` 中的数据拉取到 `buf` 缓冲区。

```
private synchronized int readWithBounceBuffer(ByteBuffer buf,
        boolean canSkipChecksum) throws IOException {
    int total = 0;
    // 调用 drainDataBuf(), 将 dataBuf 缓冲区中的数据写入 buf
    int bb = drainDataBuf(buf);
    if (bb >= 0) {
        total += bb;
        if (buf.remaining() == 0) return total;
    }
    boolean eof = true, done = false;
    do {
        // 如果 buf 的空间足够大，并且输入流游标在 chunk 边界上，则直接从 IO 流中将数据写入 buf
        if (buf.isDirect() && (buf.remaining() >= maxReadaheadLength)
            && ((dataPos % bytesPerChecksum) == 0)) {
            int oldLimit = buf.limit();
```



```

int nRead;
try {
    buf.limit(buf.position() + maxReadaheadLength);
    nRead = fillBuffer(buf, canSkipChecksum);
} finally {
    buf.limit(oldLimit);
}
if (nRead < maxReadaheadLength) {
    done = true;
}
if (nRead > 0) {
    eof = false;
}
total += nRead;
} else {
    // 否则, 将数据读入 dataBuf 缓存
    if (fillDataBuf(canSkipChecksum)) {
        done = true;
    }
    // 然后将 dataBuf 中的数据导入 buf
    bb = drainDataBuf(buf);
    if (bb >= 0) {
        eof = false;
        total += bb;
    }
}
} while ((!done) && (buf.remaining() > 0));
return (eof && total == 0) ? -1 : total;
}

```

这个设计非常的巧妙, 为什么读取操作都要先经过 `dataBuf` 缓存呢? 因为 `dataBuf` 中的数据始终保持为校验块 (chunk) 的整数倍。当循环调用 `readWithBounceBuffer()` 方法时, 第一次调用 `drainDataBuf()` 方法会确保当前输入流的游标 `pos` 定位到校验块的边界上。接下来的读取都可以以校验块的整数倍 (`maxReadaheadLength`) 读取。请参考图 5-11 所示的示例, 这里每一个 chunk 大小为 64MB, `maxReadaheadLength` 为 3 个 chunk 大小, `dataBuf` 缓冲区也为 3 个 chunk 大小, 读取数据的 `buf` 大小为 480MB。下面我们模拟 `readWithBounceBuffer()` 方法的执行过程。

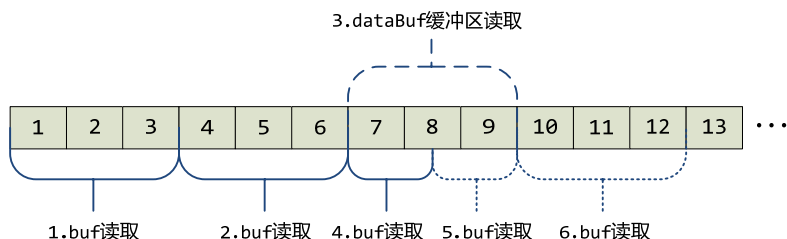


图 5-11 readWithBounceBuffer()示例

- 由于 buf 的空间足够大，并且输入流游标在 chunk 边界上，直接调用 fillBuffer()方法将 IO 流中的 3 个 chunk 读入 buf。
- 同第 1 次读取，直接将 IO 流中的 3 个 chunk 读入 buf。
- 由于 buf 的空间不足，只有 98MB，小于 maxReadaheadLength 的 192MB，则调用 fillDataBuf()方法将 3 个 chunk 写入 dataBuf 缓存。
- 从 dataBuf 中取出 98MB 数据，写入 buf 中。至此，第一次 readWithBounceBuffer()操作完成。480MB 数据读入成功。
- 第二次 readWithBounceBuffer()操作。由于 dataBuf 中还有数据，先从 dataBuf 中读取 98MB 数据到 buf 中。读取完成后，IO 流游标又回到了 chunk 边界上。
- buf 的空间足够大，并且输入流游标在 chunk 边界上，直接调用 fillBuffer()方法将 IO 流中的数据读入 buf。

3. HasEnhancedByteBufferAccess.read()

DFSInputStream 实现了 HasEnhancedByteBufferAccess 接口的 read()方法，提供了以零拷贝模式读取数据块的功能。本节主要介绍这个接口方法的具体实现。

DFSInputStream.read()首先调用 tryReadZeroCopy()方法尝试以零拷贝模式读取数据块，如果当前配置或者数据块的状态不支持零拷贝，则调用 ByteBufferUtil.fallbackRead()退化为一个普通的读取操作。

```
public synchronized ByteBuffer read(ByteBufferPool bufferPool,
    int maxLength, EnumSet<ReadOption> opts)
    throws IOException, UnsupportedOperationException {
    // ...
    ByteBuffer buffer = null;
    // 首先尝试零拷贝模式
    if (dfsClient.getConf().shortCircuitMmapEnabled) {
        buffer = tryReadZeroCopy(maxLength, opts);
    }
    if (buffer != null) {
        return buffer;
    }
    // 如果零拷贝读取不成功，则退化为一个普通的读取
    buffer = ByteBufferUtil.fallbackRead(this, bufferPool, maxLength);
    if (buffer != null) {
        extendedReadBuffers.put(buffer, bufferPool);
    }
    return buffer;
}
```

(1) tryReadZeroCopy()

在传统的文件 IO 操作中，都是调用操作系统提供的系统调用函数 read()或 write()来执行读写操作的，此时调用此函数的进程（在 Java 中即 java 进程）会由用户态切换到内核态，然后操作系统的内核代码负责将相应的文件数据读取到内核的 IO 缓冲区，最后再把数据从内核

IO 缓冲区拷贝到进程的私有地址空间中，这样便完成了一次 IO 操作。

`tryReadZeroCopy()`方法使用了内存映射文件的读取方式。内存映射文件和标准 IO 操作最大的不同是并不需要将数据读取到操作系统的内核缓冲区，而是直接将进程私有地址空间中的一部分区域与文件对象建立起映射关系，就好像直接从内存中读写文件一样，减少了 IO 的拷贝次数，提高了文件的读写速度。

Java 提供了三种内存映射模式，即：只读（`readonly`）、读写（`read_write`）、专用（`private`）。对于只读模式来说，如果程序试图进行写操作，则会抛出 `ReadOnlyBufferException` 异常；对于读写模式来说，如果程序通过内存映射文件的方式写或修改文件内容，则修改内容会立刻反映到磁盘文件中，如果另一个进程共享了同一个映射文件，也会立即看到变化；专用模式采用的是操作系统的“写时拷贝”原则，即在没有发生写操作的情况下，多个进程之间都是共享文件的同一块物理内存的（进程各自的虚拟地址指向同一片物理地址），一旦某个进程进行写操作，就会把受影响的文件数据单独拷贝一份到进程的私有缓冲区中，不会反映到物理文件中。在 `tryReadZeroCopy()`方法中使用的是只读模式。

对于数据文件的读取，内存映射读取大大提高了性能，这种模式值得积累。

了解了零拷贝读取的概念之后，我们来看 `tryReadZeroCopy()`方法的代码实现。`tryReadZeroCopy()`会通过调用序列 `BlockReader.getClientMmap()-> ShortCircuitReplica.getOrCreateClientMmap()-> ShortCircuitCache.getOrCreateClientMmap()-> ShortCircuitReplica.loadMmapInternal()` 获取 `MappedByteBuffer` 对象，也就是数据块文件在内存中的映射对象。`ShortCircuitReplica.loadMmapInternal()`的代码如下：

```
MappedByteBuffer loadMmapInternal() {
    try {
        FileChannel channel = dataStream.getChannel();
        // 调用 java.nio.Channel.map() 方法创建文件的内存映射
        MappedByteBuffer mmap = channel.map(MapMode.READ_ONLY, 0,
            Math.min(Integer.MAX_VALUE, channel.size()));
        return mmap;
    } catch (IOException e) {
        return null;
    } catch (RuntimeException e) {
        return null;
    }
}
```

`tryReadZeroCopy()`方法首先通过上面描述的 `getClientMmap()`方法获取数据块文件的内存映射对象 `clientMmap`，`clientMmap` 对象中保存了一个 `MappedByteBuffer` 对象，也就是数据块文件在内存中的映射缓冲区，`tryReadZeroCopy()`会通过 `clientMmap` 获取这个 `MappedByteBuffer` 对象并将这个对象返回。接下来用户代码就可以从 `MappedByteBuffer` 这个对象读取数据了。这里要特别注意的是，Java 的 `ByteBuffer` 只支持 2GB 以下的空间，因为 `ByteBuffer` 使用 4 字节的地址空间，所以需要对加载数据的大小进行判断，超过 2GB 不予加载。`tryReadZeroCopy()`方法的代码如下：

Hadoop 2.X HDFS 源码剖析

```
private synchronized ByteBuffer tryReadZeroCopy(int maxLength,
    EnumSet<ReadOption> opts) throws IOException {
    final long curPos = pos;
    final long curEnd = blockEnd;
    final long blockStartInFile = currentLocatedBlock.getStartOffset();
    final long blockPos = curPos - blockStartInFile;

    // 首先确保读取是在同一个数据块之内
    long length63;
    if ((curPos + maxLength) <= (curEnd + 1)) {
        length63 = maxLength;
    } else {
        length63 = 1 + curEnd - curPos;
        if (length63 <= 0) {
            return null;
        }
    }
    // 确保读取映射数据没有超过 2GB
    int length;
    if (blockPos + length63 <= Integer.MAX_VALUE) {
        length = (int)length63;
    } else {
        long length31 = Integer.MAX_VALUE - blockPos;

        return null;
    }
    length = (int)length31;
}
// 调用 blockReader.getClientMmap() 将文件映射到内存中, 并返回 ClientMmap 对象。这个对象当中
// 包含了 MappedByteBuffer 对象
final ClientMmap clientMmap = blockReader.getClientMmap(opts);
if (clientMmap == null) {
    return null;
}
boolean success = false;
ByteBuffer buffer;
try {
    seek(curPos + length);
    // 将内存映射缓冲区返回, 在缓冲区中是数据块文件的数据
    buffer = clientMmap.getMappedByteBuffer().asReadOnlyBuffer();
    buffer.position((int)blockPos);
    buffer.limit((int)(blockPos + length));
    extendedReadBuffers.put(buffer, clientMmap);
    readStatistics.addZeroCopyBytes(length);
    success = true;
} finally {
    if (!success) {
        IOUtils.closeQuietly(clientMmap);
    }
}
```

```

    }
}
return buffer;
}

```

(2) ByteBufferUtil fallbackRead()

当 read()方法执行零拷贝读操作失败后，会调用 ByteBufferUtil fallbackRead()退化为一个普通的读操作。ByteBufferUtil fallbackRead()方法非常简单，判断传入参数的 InputStream (DFSInputStream) 是否支持 ByteBuffer Read (实现了 ByteBufferReadable 接口)。如果支持则直接将数据读取至 ByteBuffer 中，否则读取到 ByteBuffer.array()字节数组中。这个方法在 InputStream.read()小节中已经介绍过了。

```

public static ByteBuffer fallbackRead(
    InputStream stream, ByteBufferPool bufferPool, int maxLength)
    throws IOException {
    if (bufferPool == null) {
        // 抛出异常
    }
    // 判断 stream 是否支持将数据读入 ByteBuffer
    boolean useDirect = streamHasByteBufferRead(stream);
    // 调用 ByteBufferPool 构造一个 ByteBuffer
    ByteBuffer buffer = bufferPool.getBuffer(useDirect, maxLength);
    if (buffer == null) {
        // ByteBufferPool 无法构造 ByteBuffer
    }
    Preconditions.checkState(buffer.capacity() > 0);
    Preconditions.checkState(buffer.isDirect() == useDirect);
    maxLength = Math.min(maxLength, buffer.capacity());
    boolean success = false;
    try {
        if (useDirect) {
            buffer.clear();
            buffer.limit(maxLength);
            ByteBufferReadable readable = (ByteBufferReadable)stream;
            int totalRead = 0;
            while (true) {
                if (totalRead >= maxLength) {
                    success = true;
                    break;
                }
                // 直接调用 stream 上支持 ByteBuffer Read 的函数
                int nRead = readable.read(buffer);
                if (nRead < 0) {
                    if (totalRead > 0) {
                        success = true;
                    }
                    break;
                }
            }
        }
    }
}

```

```
        }
        totalRead += nRead;
    }
    buffer.flip();
} else {
    buffer.clear();
    // 调用 InputStream.read(byte[]) 方法
    int nRead = stream.read(buffer.array(),
        buffer.arrayOffset(), maxLength);
    if (nRead >= 0) {
        buffer.limit(nRead);
        success = true;
    }
}
} finally {
    if (!success) {
        bufferPool.putBuffer(buffer);
        buffer = null;
    }
}
return buffer;
}
```

4. 关闭输入流

用户代码读取完所有数据之后，就会调用 `DFSInputStream.close()` 方法关闭输入流。`close()` 方法的实现也非常简单，它首先检查 `DFSClient` 是否处于运行状态，然后关闭读取过程中可能使用过的 `ByteBuffer`，最后调用 `BlockReader.close()` 关闭当前输入流底层的 `BlockReader`。

```
public synchronized void close() throws IOException {
    if (closed) {
        return;
    }
    dfsClient.checkOpen();
    // 关闭读取过程中使用的 ByteBuffer
    if (!extendedReadBuffers.isEmpty()) {
        final StringBuilder builder = new StringBuilder();
        extendedReadBuffers.visitAll(new IdentityHashStore.Visitor<ByteBuffer, Object>() {
            private String prefix = "";
            @Override
            public void accept(ByteBuffer k, Object v) {
                builder.append(prefix).append(k);
                prefix = ", ";
            }
        });
    }
    // 关闭 BlockReader 对象
    if (blockReader != null) {
        blockReader.close();
    }
}
```

```

    blockReader = null;
}
// 调用父类的 close() 方法
super.close();
closed = true;
}

```

5.3 文件短路读操作

Hadoop 的一个重要思想就是移动计算，而不是移动数据。这种设计方式使得客户端常常与数据块所在的 **Datanode** 在同一台机器上，那么当 **DFSClient** 读取一个本地数据块时，就会出现本地读取（**LocalRead**）操作。在 HDFS 早期版本中，本地读取和远程读取的实现是一样的，如图 5-12 所示，客户端通过 TCP 套接字连接 **Datanode**，并通过 **DataTransferProtocol** 传输数据。这种方式很简单，但是有一些不好的地方，例如 **Datanode** 需要为每个读取数据块的客户端都维持一个线程和 TCP 套接字。内核中 TCP 协议是有开销的，**DataTransferProtocol** 本身也有开销，因此这种实现方式有值得优化的地方。

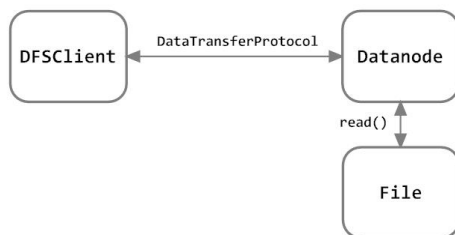


图 5-12 网路读操作示意图

既然客户端和 **Datanode** 在同一台机器上，那么 **DFSClient** 可不可以不通过 **Datanode** 而直接读取磁盘上的数据呢？短路读取就是这种思想的实现，目前 HDFS 提供了两种短路读取方案。

1. FS-2246

Datanode 将所有数据路径权限开放给客户端，当执行一个本地读取时，客户端直接从本地磁盘的数据路径读取数据。但这种实现方式带来了安全问题，客户端用户可以直接浏览所有数据，可见这并不是一个很好的选择。HDFS-2246 实现的短路读取模式如图 5-13 所示。

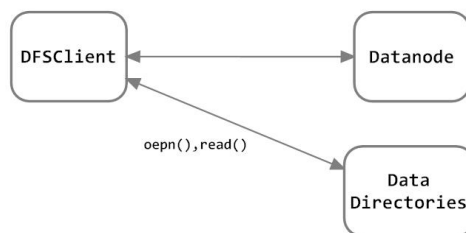


图 5-13 HDFS-2246 短路读操作示意图

2. HDFS-347

UNIX 提供了一种 UNIX Domain Socket 进程间通信方式，它使得同一台机器上的两个进程能以 Socket 的方式通信，并且还可以在进程间传递文件描述符。

HDFS-347 使用该机制实现了安全的本地短路读取，如图 5-14 所示。客户端向 Datanode 请求数据时，Datanode 会打开块文件和校验和文件，将这两个文件的文件描述符直接传给客户端，而不是将路径传给客户端。客户端接收到这两个文件的文件描述符之后，就可以直接打开文件读取数据了，也就绕过了 Datanode 进程的转发，提高了读取效率。因为文件描述符是只读的，所以客户端不能修改该文件。同时，由于客户端自身无法访问数据块文件所在的目录，所以它也就不能访问其他不该访问的数据了，保证了读取的安全性。HDFS 2.X 采取的就是 HDFS-347 的设计实现短路读取功能的。

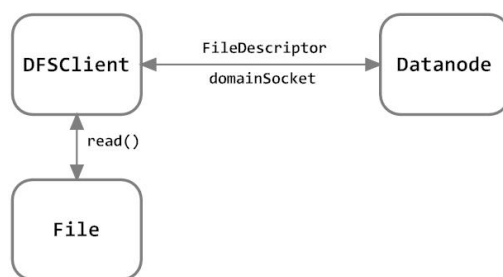


图 5-14 HDFS-347 短路读操作示意图

5.3.1 短路读共享内存

了解了短路读取的概念之后，我们看一下 HDFS 是如何实现这种模式的。在 DFSClient 中，使用 ShortCircuitReplica 类封装可以进行短路读取的副本。ShortCircuitReplica 对象中包含了短路读取副本的数据块文件输入流、校验文件输入流、短路读取副本在共享内存中的槽位（slot）以及副本的引用次数等信息。DFSClient 会持有一个 ShortCircuitCache 对象缓存并管理所有的 ShortCircuitReplica 对象，DFSClient 从 ShortCircuitCache 获得了 ShortCircuitReplica 的引用之后，就可以构造 BlockReaderLocal 对象进行本地读取操作了。

如图 5-15 所示，当 DFSClient 和 Datanode 在同一台机器上时，需要一个共享内存段来维护所有短路读取副本的状态，共享内存段中会有很多个槽位，每个槽位都记录了一个短路读取副本的信息，例如当前副本是否有效、锚（anchor）的次数等。这里我们解释一下锚的概念。当 Datanode 将一个数据块副本缓存到内存中时，会将这个数据块副本设置为可锚（anchorable）状态，也就是在共享内存中该副本对应的槽位上设置可锚状态位（请参考 Slot 类实现小节中的分析）。当一个副本被设置为可锚状态之后，DFSClient 的 BlockReaderLocal 对象读取该副本时就不再需要校验操作了（因为缓存中的副本已经执行过校验操作，请参考第 4 章的 FSDataSetImpl 实现小节），并且输入流可以通过零拷贝模式读取这个副本（请参考 DFSInputStream 实现小节）。每当客户端进行这两种读取操作时，都需要在副本对应的槽位上

添加一个锚计数，只有副本的锚计数为零时，Datanode 才可以从缓存中删除这个副本。可以看到，共享内存以及槽位机制很好地在 Datanode 进程和 DFSClnt 进程间同步了副本的状态，保证了 Datanode 缓存操作以及 DFSClnt 读取副本操作的正确性。

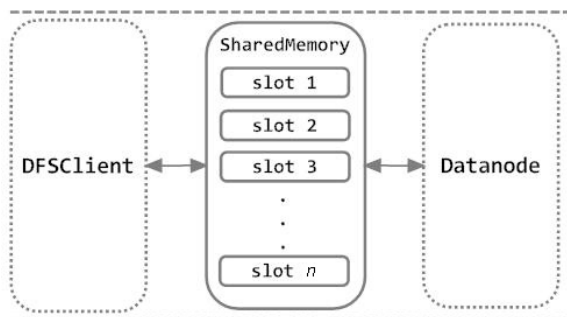


图 5-15 共享内存段的结构

如图 5-16 所示，共享内存机制是由 DFSClnt 和 Datanode 对同一个文件执行内存映射操作实现的。因为 MappedByteBuffer 对象能让内存与物理文件的数据实时同步，所以 DFSClnt 和 Datanode 进程会通过中间文件来交换数据，中间文件使得两个进程的内存区域得到及时的同步。DFSClnt 和 Datanode 之间可能会有多段共享内存，所以 DFSClnt 定义了 DFSClntShm 类抽象 DFSClnt 侧的一段共享内存，定义了 DFSClntShmManager 类管理所有的 DFSClntShm 对象；而 Datanode 则定义了 RegisteredShm 类抽象 Datanode 侧的一段共享内存，同时定义了 ShortCircuitRegistry 类管理所有 Datanode 侧的共享内存。

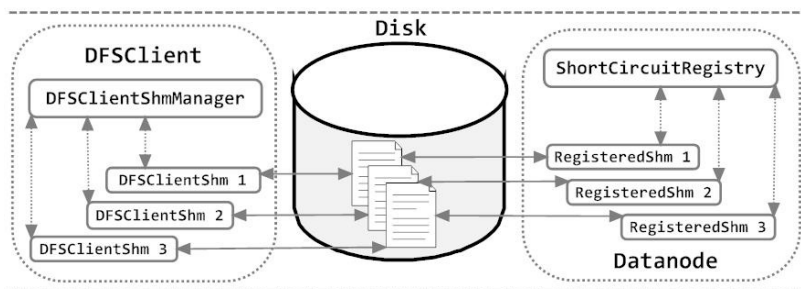


图 5-16 共享内存机制示意图

DFSClnt 会调用 DataTransferProtocol.requestShortCircuitShm 接口与 Datanode 协商创建一段共享内存，共享内存创建成功后，DFSClnt 和 Datanode 会分别构造 DFSClntShm 以及 RegisteredShm 对象维护这段共享内存。如图 5-17 所示，共享内存中的文件映射数据是实时同步的，它保存了所有槽位的二进制信息。但是映射数据中二进制的槽位信息并不便于操作，所以 DFSClntShm 和 RegisteredShm 会构造一个 Slot 对象操作映射数据中的一个槽位，同时各自定义了集合字段保存所有的 Slot 对象。这里需要特别注意的是，Slot 对象会由 DFSClntShm 和 RegisteredShm 分别构造并保存在各自的集合字段中，所以 DFSClntShm 和

RegisteredShm 之间需要同步 Slot 对象的创建和删除操作，以保证 DFSClietShm 和 RegisteredShm 保存的 Slot 对象信息是完全同步的。DataTransferProtocol 接口就提供了 requestShortCircuitFds()以及 releaseShortCircuitFds()方法同步 Slot 对象的创建和删除操作，我们在下一个小节中介绍 DataTransferProtocol 的实现。

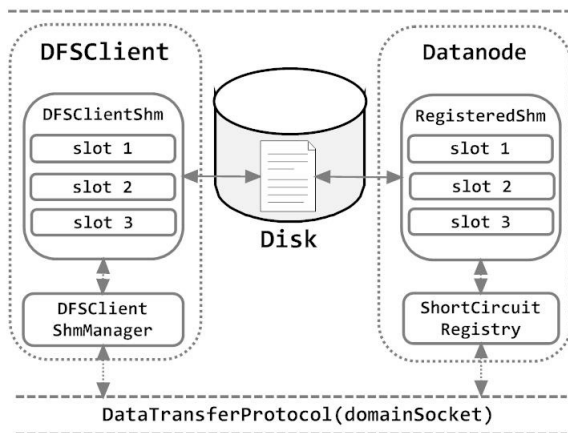


图 5-17 Slot 状态同步示意图

5.3.2 DataTransferProtocol

在第 4 章中我们介绍过流式接口 DataTransferProtocol，如图 5-18 所示，DataTransferProtocol 定义了 requestShortCircuitShm()、requestShortCircuitFds()以及 releaseShortCircuitFds()三个接口方法同步 Datanode 和 DFSCliet 对共享内存的操作。

```

DataTransferProtocol
@ readBlock(ExtendedBlock, Token<BlockTokenIdentifier>, String, long, long, boolean, CachingStrategy) void
@ writeBlock(ExtendedBlock, StorageType, Token<BlockTokenIdentifier>, String, DatanodeInfo[], StorageType[], DatanodeInfo, BlockCons void
@ transferBlock(ExtendedBlock, Token<BlockTokenIdentifier>, String, DatanodeInfo[], StorageType[]) void
@ requestShortCircuitFds(ExtendedBlock, Token<BlockTokenIdentifier>, SlotId, int) void
@ releaseShortCircuitFds(SlotId) void
@ requestShortCircuitShm(String) void
@ replaceBlock(ExtendedBlock, StorageType, Token<BlockTokenIdentifier>, String, DatanodeInfo) void
@ copyBlock(ExtendedBlock, Token<BlockTokenIdentifier>) void
@ blockChecksum(ExtendedBlock, Token<BlockTokenIdentifier>) void

```

图 5-18 DataTransferProtocol 接口定义

需要注意的是，DataTransferProtocol 底层是基于 Socket 流的，而当 DFSCliet 和 Datanode 在同一台物理机器上时，DataTransferProtocol 底层的 Socket 将会是 DomainSocket，使用 DomainSocket 的 DataTransferProtocol 可以在 Socket 流中传递文件描述符。

本节的内容应该安排在第 4 章，但为了将短路读取作为一个整体介绍，所以把这部分内容放在了本章。本节涉及的 DataTransferProtocol 以及 DataXceiver 的相关内容请读者参考第 4 章的流式接口小节，下面我们就依次看一下 requestShortCircuitShm()、requestShortCircuitFds()

以及 `releaseShortCircuitFds()` 方法的实现。

1. `requestShortCircuitShm()`

DFSClient 在执行任何短路读取操作之前，需要先申请一段共享内存保存短路读取副本的状态。DFSClient 会调用 `DataTransferProtocol.requestShortCircuitShm()` 方法向 Datanode 发起申请共享内存的请求，Datanode 的 `DataXceiver.requestShortCircuitShm()` 方法会响应这个请求（请参考第4章的流式接口小节）。

如下代码所示，`DataXceiver.requestShortCircuitShm()` 会调用 `ShortCircuitRegistry.createNewMemorySegment()` 方法创建共享内存段，`createNewMemorySegment()` 方法会将共享内存文件映射到 Datanode 的内存中，然后构造 `RegisteredShm` 类管理这段共享内存（请参考 `ShortCircuitRegistry` 类小节中的分析）。之后 `DataXceiver.requestShortCircuitShm()` 方法会调用 `sendShmSuccessResponse()` 方法将共享内存文件的文件描述符通过 `domainSocket` 发回客户端。

```
public void requestShortCircuitShm(String clientName) throws IOException {
    NewShmInfo shmInfo = null;
    boolean success = false;
    DomainSocket sock = peer.getDomainSocket(); // 获取底层 DomainSocket 对象
    try {
        if (sock == null) { // 如果 DataTransferProtocol 底层不是 DomainSocket，则发回异常
            sendShmErrorResponse(ERROR_INVALID, "Bad request from " +
                peer + ": must request a shared " +
                "memory segment over a UNIX domain socket.");
            return;
        }
        try {
            // 调用 ShortCircuitRegistry.createNewMemorySegment() 方法创建共享内存段
            shmInfo = datanode.shortCircuitRegistry.
                createNewMemorySegment(clientName, sock);
            releaseSocket();
        } catch (UnsupportedOperationException e) { // 抛出异常，则响应异常消息
            sendShmErrorResponse(ERROR_UNSUPPORTED,
                "This datanode has not been configured to support " +
                "short-circuit shared memory segments.");
            return;
        } catch (IOException e) { // 抛出异常，则响应异常消息
            sendShmErrorResponse(ERROR,
                "Failed to create shared file descriptor: " + e.getMessage());
            return;
        }
        // 调用 sendShmSuccessResponse() 方法将共享内存文件的文件描述符发回客户端
        sendShmSuccessResponse(sock, shmInfo);
        success = true;
    } finally {
        if ((!success) && (peer == null)) {
            IOUtils.cleanup(null, sock);
        }
    }
}
```

```

    }
    IOUtils.cleanup(null, shmInfo);
}
}

```

DFSClient 的 DfsClientShmManager 对象从 domainSocket 接收了共享内存文件的文件描述符后，会打开共享内存文件并将该文件映射到 DFSClient 的内存中，之后创建 DfsClientShm 对象管理这段共享内存（请参考 DfsClientShmManager 类小节中的分析），并将这个 DfsClientShm 对象保存在 DfsClientShmManager 的对应字段中。

2. requestShortCircuitFds()

DFSClient 只有构造了 ShortCircuitReplica 对象才可以执行短路读取操作。DFSClient 在构造 ShortCircuitReplica 对象前，需要先调用 DataTransferProtocol.requestShortCircuitFds() 方法向 Datanode 申请数据块文件以及校验和文件的文件描述符，并且同步共享内存中 Slot 对象的状态。客户端在调用 requestShortCircuitFds() 方法前，会在 DFSClient 侧的共享内存中为数据块申请一个槽位并构造 Slot 对象，之后在调用 requestShortCircuitFds() 方法时传入该槽位的信息，Datanode 的 DataXceiver.requestShortCircuitFds() 方法会响应这个请求。

如下代码所示，DataXceiver.requestShortCircuitFds() 方法首先会调用 ShortCircuitRegistry.registerSlot() 方法在 Datanode 侧的共享内存中创建槽位信息对应的 Slot 对象。然后 DataXceiver.requestShortCircuitFds() 方法会调用 DataNode.requestShortCircuitFdsForRead() 方法获取数据块文件以及校验和文件的文件描述符，并通过 DomainSocket 发送给客户端。

```

public void requestShortCircuitFds(final ExtendedBlock blk,
    final Token<BlockTokenIdentifier> token,
    SlotId slotId, int maxVersion) throws IOException {
    BlockOpResponseProto.Builder bld = BlockOpResponseProto.newBuilder();
    FileInputStream fis[] = null;
    try {
        if (peer.getDomainSocket() == null) { // 如果底层不是 DomainSocket，则抛出异常
            throw new IOException("You cannot pass file descriptors over " +
                "anything but a UNIX domain socket.");
        }
        if (slotId != null) {
            boolean isCached = datanode.data.
                isCached(blk.getBlockPoolId(), blk.getBlockId());
            // 调用 ShortCircuitRegistry.registerSlot() 方法在 Datanode 的共享内存中注册这个 Slot 对象
            datanode.shortCircuitRegistry.registerSlot(
                ExtendedBlockId.fromExtendedBlock(blk), slotId, isCached);
        }
        try {
            // 获取数据块文件以及校验和文件的文件描述符
            fis = datanode.requestShortCircuitFdsForRead(blk, token, maxVersion);
        } finally {
            if ((fis == null) && (slotId != null)) {
                datanode.shortCircuitRegistry.unregisterSlot(slotId);
            }
        }
    }
}

```

```

    }
    // 构造响应消息
    bld.setStatus(SUCCESS);
    bld.setShortCircuitAccessVersion(DataNode.CURRENT_BLOCK_FORMAT_VERSION);
} catch (ShortCircuitFdsVersionException e) {
    // ...处理异常
} ... // 处理异常
try {
    // 发回成功的响应消息
    bld.build().writeDelimitedTo(socketOut);
    if (fis != null) {
        FileDescriptor fds[] = new FileDescriptor[fis.length];
        for (int i = 0; i < fds.length; i++) {
            fds[i] = fis[i].getFD();
        }
        byte buf[] = new byte[] { (byte)0 };
        // 通过 DomainSocket 将数据块文件和校验和文件的文件描述符发送给客户端
        peer.getDomainSocket().
            sendFileDescriptors(fds, buf, 0, buf.length);
    }
} finally {
    if (fis != null) {
        IOUtils.cleanup(LOG, fis);
    }
}
}

```

DFSClient 的 BlockReaderFactory 对象成功地从 domainSocket 接收了数据块文件和校验和文件的文件描述符之后，就可以初始化数据块文件和校验和文件的输入流并构造 ShortCircuitReplica 对象了。

3. releaseShortCircuitFds()

当客户端完成了对副本的短路读操作时，就需要关闭 ShortCircuitReplica 对象，同时调用 DataTransferProtocol.releaseShortCircuitFds()方法删除 Datanode 侧共享内存中的 Slot 对象，这个请求是由 DataXceiver.releaseShortCircuitFds()方法响应的。

如下代码所示，DataXceiver.releaseShortCircuitFds()方法会从 Datanode 侧的共享内存中释放这个 Slot 对象，之后发回响应消息。DFSClient 接收到响应消息后，也会在自己的共享内存中释放对应的 Slot 对象，完成整个读入操作。

```

public void releaseShortCircuitFds(SlotId slotId) throws IOException {
    boolean success = false;
    try {
        String error;
        Status status;
        try {
            // 释放共享内存中的槽位
            datanode.shortCircuitRegistry.unregisterSlot(slotId);

```

```
        error = null;
        status = Status.SUCCESS;
    } catch (UnsupportedOperationException e) {
        error = "unsupported operation";
        status = Status.ERROR_UNSUPPORTED;
    } catch (Throwable e) {
        error = e.getMessage();
        status = Status.ERROR_INVALID;
    }
    // 构造响应消息
    ReleaseShortCircuitAccessResponseProto.Builder bld =
        ReleaseShortCircuitAccessResponseProto.newBuilder();
    bld.setStatus(status);
    if (error != null) {
        bld.setError(error);
    }
    // 发回响应消息
    bld.build().writeDelimitedTo(socketOut);
    success = true;
} finally {
}
}
```

5.3.3 DFSClient 短路读操作流程

我们在 DFSInputStream 实现小节中介绍了，DFSInputStream 在执行数据块副本的本地短路读操作时，需要构造一个 BlockReaderLocal 对象。BlockReaderLocal 对象的底层输入流是通过 ShortCircuitReplica 对象获取的，因为 ShortCircuitReplica 对象保存了副本的数据块文件和校验和文件的文件描述符，ShortCircuitReplica 对象会通过这两个文件描述符构造输入流。

图 5-19 给出了客户端构造 ShortCircuitReplica 对象的流程。客户端首先会调用 ShortCircuitCache.fetchAndCreate()方法尝试从缓存中获取 ShortCircuitReplica 对象，如果缓存中没有这个对象，fetchAndCreate()方法会调用 create()方法创建一个新的 ShortCircuitReplica 对象，并在创建成功之后将这个对象放入 ShortCircuitCache 缓存中。create()方法首先会调用 DFSClientShmManager.allocSlot()方法尝试在已有的共享内存中获取一个槽位，如果在 DFSClientShmManager 管理的所有共享内存中都没有槽位了，allocSlot()方法会调用 DataTransferProtocol.requestShortCircuitShm()方法申请一段新的共享内存，并构造一个 DfsClientShm 对象管理这段共享内存，然后 allocSlot()方法会在这段共享内存中获取一个存放当前副本状态的槽位并返回。处理好槽位后，create()方法会调用 DataTransferProtocol.requestShortCircuitFds()方法获取副本的文件描述符并构建 IO 流，然后构建 ShortCircuitReplica 对象并返回。

上面的流程中涉及如下几个比较重要的类，后面我们会一一介绍这些类的实现。

- ShortCircuitShm: 抽象了一段共享内存。
- Slot: 一段共享内存中会有多个槽位，每个槽位都对应一个 Slot 对象。每个槽位中都

保存了短路读副本的信息，包括当前槽位是否有效，以及当前副本锚的次数。当 **Datanode** 在内存中缓存了一个数据块副本时（通过 **mlock()** 系统调用），该副本对应的槽位会被设置为可锚（**Anchorable**）状态，可锚状态的数据块在读取时可以不用进行校验，同时可锚状态的数据块可以进行真正的零拷贝读取。这里要特别注意的是，当客户端对这个副本执行一次免校验读取或者零拷贝读取时，需要对副本的锚次数加 1。同时当 **Datanode** 在内存中 **munlock()** 一个数据块时，则需要首先等待客户端的免校验读取以及零拷贝读取完成，也就是锚次数为零时，才可以执行 **munlock()** 数据块的操作。

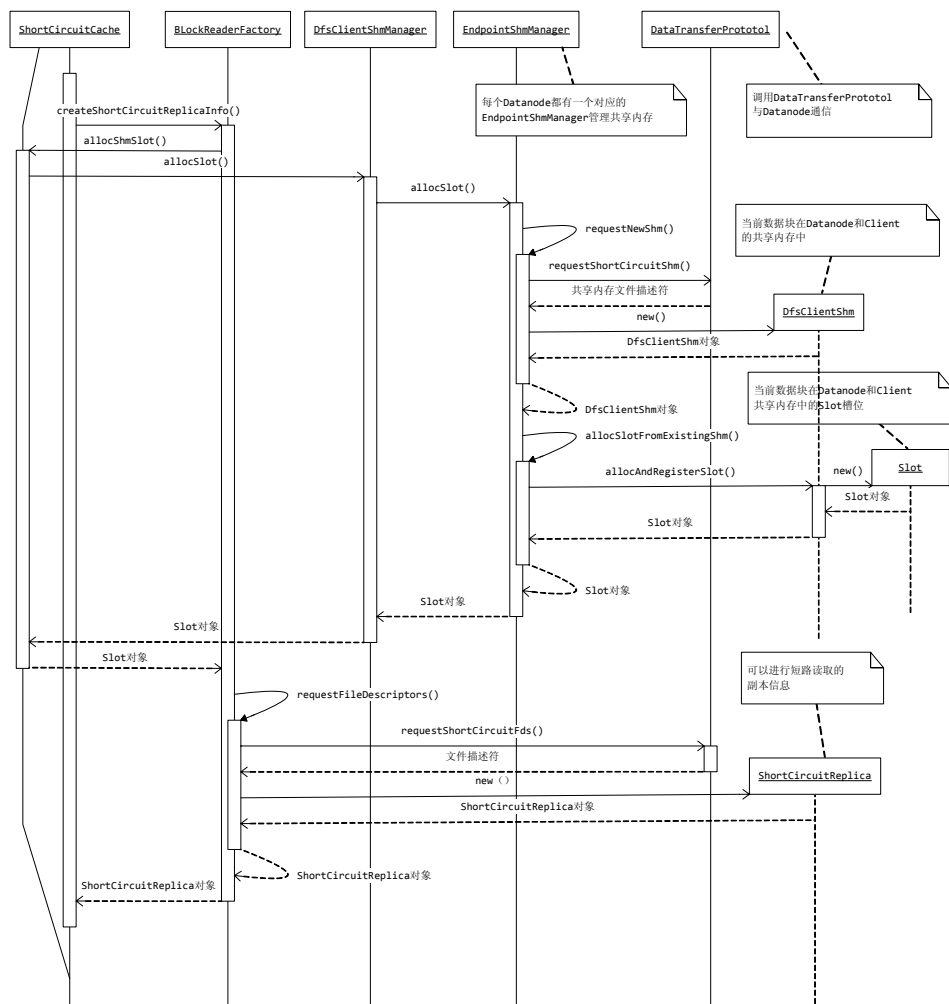


图 5-19 短路读操作流程图

- **DfsClientShm**: **ShortCircuitShm** 的子类，用来抽象 **DFSCClient** 侧的一段共享内存。

- **DfsClientShmManager**: 管理 DFSClnt 侧的所有共享内存。如上面的短路读操作流程图所示, 客户端使用这个类申请新的共享内存段、在共享内存段上申请新的槽位等操作。这个类在 Datanode 上的对应类是 ShortCircuitRegistry。
- **ShortCircuitReplica**: 用来保存短路读取数据块的信息, 包括文件描述符、文件的 IO 流等。
- **ShortCircuitCache**: 在 DFSClnt 端缓存已有的 ShortCircuitReplica 对象。

1. ShortCircuitShm 类

ShortCircuitShm 类用来抽象短路读取时用到的一段共享内存, 这段共享内存中会有多个槽位, 每个槽位都保存了一个短路读副本的信息。ShortCircuitShm 类结构如图 5-20 所示。



图 5-20 ShortCircuitShm 类结构

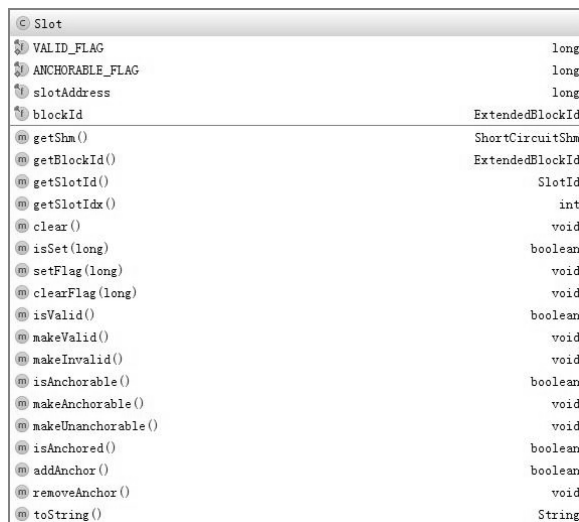
ShortCircuitShm 定义了如下 4 个内部类。

- **ShmId**: 用来唯一标识一个 ShortCircuitShm, 有两个 long 类型的 id 字段, 是随机产生的。
- **Slot**: 用于描述共享内存中的一个槽位, 每个槽位都用来追踪一个短路数据块的状态。由于是在共享内存中, 这个状态可以由 Datanode 进程修改, 也可以由 DFSClnt 修改。
- **SlotId**: 用来唯一标识一个 Slot。
- **SlotIterator**: 迭代器类型, 用来遍历当前 ShortCircuitShm 中保存的所有 Slot 对象。

(1) Slot 类实现

Slot 类是 ShortCircuitShm 中最重要的一个内部类, 它抽象了共享内存中的一个槽位, 并

提供了操作内存映射文件对应槽位数据的方法。Slot 类结构如图 5-21 所示。



Field/Method	Type
VALID_FLAG	long
ANCHORABLE_FLAG	long
slotAddress	long
blockId	ExtendedBlockId
getShm()	ShortCircuitShm
getBlockId()	ExtendedBlockId
getSlotId()	SlotId
getSlotIdx()	int
clear()	void
isSet(long)	boolean
setFlag(long)	void
clearFlag(long)	void
isValid()	boolean
makeValid()	void
makeInvalid()	void
isAnchorable()	boolean
makeAnchorable()	void
makeUnanchorable()	void
isAnchored()	boolean
addAnchor()	boolean
removeAnchor()	void
toString()	String

图 5-21 Slot 类结构

Slot 类有两个字段：slotAddress，保存当前 Slot 在共享内存中的地址；blockId，当前 Slot 对应的短路读数据块 id。

Slot 类有两个标志位，即 VALID_FLAG 和 ANCHORABLE_FLAG，标志位的信息都保存在共享内存文件中。

- VALID_FLAG——用来标识当前 Slot 是否有效。DFSClient 在共享内存中新创建一个 Slot 时，会设置这个标志位。当这个 Slot 对应的副本不再有效时，Datanode 取消这个标志位。同时，当 Client 认为 Datanode 不再使用这个 Slot 通信时，也会取消这个标志位。
- ANCHORABLE_FLAG——当 Datanode 将 Slot 对应的副本通过 mlock 操作缓存到内存中时，会设置这个标志位。当这个标志位被设置时，DFSClient 的 BlockReaderLocal 读取 Slot 对应的副本时不再需要校验，并且客户端可以进行零拷贝读取。每当客户端进行这样的读取操作时，都需要在 Slot 上添加一个锚计数，只有当 Slot 没有引用的锚，也就是锚计数为零时，Datanode 才可以从缓存中移出这个数据块。

这里要注意，每个 Slot 对象在共享内存中都是以 8 字节的 long 类型存储的。我们首先看一下用于设置标志位的 setFlag() 方法实现。setFlag() 方法设置标志位使用位操作 `prev | flag`，取消标志位则使用 `prev & (~flag)`。

```
private void setFlag(long flag) {
    long prev;
    do {
        // 这里通过调用 native 方法，从内存中获取这 8 字节
        prev = unsafe.getLongVolatile(null, this.slotAddress);
```

```
// 进行与操作，如果结果不为空，则证明标志位已经设置了
if ((prev & flag) != 0) {
    return;
}
// 否则，就将当前值与 flag 进行或操作，也就添加了标志位
} while (!unsafe.compareAndSwapLong(null, this.slotAddress,
    prev, prev | flag));
}
```

学习了设置标志位的方法后，我们来看一下设置 VALID_FLAG 标志位的 makeValid()方法，以及设置 ANCHORABLE_FLAG 标志位的 makeAnchorable()方法，这两个方法都调用了 setFlag()方法。makeValid()方法是在 Slot 对象创建时调用的，也就是 Slot 对象在创建之后就是有效的。makeAnchorable()方法则是在数据块被成功放入缓存时，也就是 CachingTask 成功调用 mlock()系统调用后执行的，可锚（Anchorable）槽位对应的数据块在读取时是不需要校验的，也就是在本地读取该数据块时不需要执行校验操作，同时客户端可以使用零拷贝模式读取这个数据块。

```
public void makeValid() {
    setFlag(VALID_FLAG);
}

public void makeAnchorable() {
    setFlag(ANCHORABLE_FLAG);
}
```

Slot 类中还有一个非常重要的方法是 addAnchor()，如图 5-22 所示，当客户端通过免校验模式（缓存中的数据块不需要校验）或者零拷贝模式读取数据块时，会调用 addAnchor()方法增加这个 Slot 的锚次数，表明当前数据块正在被引用。只有当 Slot 的锚次数为零时，Datanode 才可以将该 Slot 对应的数据块从缓存中移出。

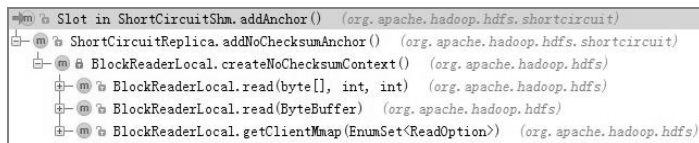


图 5-22 addAnchor()方法调用关系

如下代码所示，addAnchor()方法首先判断 Slot 是否有效并且可锚，然后判断是否有太多线程锚定了这个 Slot，如果不是则增加 Slot 对象的锚次数。

```
public boolean addAnchor() {
    long prev;
    do {
        prev = unsafe.getLongVolatile(null, this.slotAddress);
        if ((prev & VALID_FLAG) == 0) {
            // Slot 不是有效的，直接返回 false
            return false;
        }
    }
```

```

    if ((prev & ANCHORABLE_FLAG) == 0) {
        // Slot 并不是可锚的, 直接返回 false
        return false;
    }
    if ((prev & 0x7fffffff) == 0x7fffffff) {
        // 有太多线程锚定了这个 Slot
        return false;
    }
    // 否则, 增加当前 Slot 的锚次数
} while (!unsafe.compareAndSwapLong(null, this.slotAddress,
                                     prev, prev + 1));
return true;
}

```

(2) ShortCircuitShm 字段

了解了 ShortCircuitShm 的内部类之后, 我们看一下这个类中的字段。

- Slot slots[]: 用来保存共享内存中所有的 Slot 对象。
- BitSet allocatedSlots: 一个 bitmap 用来标识 slots 数组中哪些位置被占用。这里积累 BitSet 的使用。
- long baseAddress: 内存映射文件的起始地址。
- int mmapLength: 内存映射文件的长度。

(3) ShortCircuitShm 方法

ShortCircuitShm 实现共享内存是通过 mmap 映射文件的方式, 客户端和 Datanode 会打开同一个文件, 然后进行内存映射以达到共享内存的目的。我们首先看一下 ShortCircuitShm 的构造方法, 它会首先打开共享文件, 执行 mmap 内存映射操作, 然后根据共享文件的大小计算出共享内存中可以有多少个 Slot, 最后构造 slots 字段以及 allocatedSlots 字段。

```

public ShortCircuitShm(ShmId shmId, FileInputStream stream)
    throws IOException {
    this.shmId = shmId;
    this.mmapLength = getUsableLength(stream);
    // 可以看到, 这里实现共享内存是通过 mmap 方式, 打开同一个文件, 并进行内存映射
    this.baseAddress = POSIX.mmap(stream.getFD(),
        POSIX.MMAP_PROT_READ | POSIX.MMAP_PROT_WRITE, true, mmapLength);
    // 每个 Slot 是 8 字节, 这里通过文件的大小计算出当前共享内存中有多少个 Slot
    this.slots = new Slot[mmapLength / BYTES_PER_SLOT];
    this.allocatedSlots = new BitSet(slots.length);
}

```

ShortCircuitShm 提供了 allocAndRegisterSlot()方法在当前共享内存中注册一个 Slot 对象, allocAndRegisterSlot()方法首先从 slots 数组中获取一个空闲的位置, 然后构造 Slot 对象, 将这个 Slot 对象放入 slots 数组中, 最后返回新构造的 Slot 对象。allocAndRegisterSlot()方法的代码如下:

```

synchronized public final Slot allocAndRegisterSlot(

```

```
    ExtendedBlockId blockId) {
    // 获取第一个空闲的位置, 这里使用了 BitSet (注意积累 BitSet 的使用)
    int idx = allocatedSlots.nextClearBit(0);
    allocatedSlots.set(idx, true);
    // 构造 Slot 对象, 然后放入 slots 字段中
    Slot slot = new Slot(calculateSlotAddress(idx), blockId);
    slot.makeValid();
    slots[idx] = slot;
    return slot;
}
```

`unregisterSlot()`方法则用在对应的解注册流程中, 用于从 `ShortCircuitShm` 移除一个 `Slot` 对象。

```
synchronized public final void unregisterSlot(int slotIdx) {
    Preconditions.checkState(allocatedSlots.get(slotIdx),
        "tried to unregister slot " + slotIdx + ", which was not registered.");
    allocatedSlots.set(slotIdx, false);
    // 将 slots 数组中对应的槽位设置为 null
    slots[slotIdx] = null;
}
```

当共享内存中所有的槽位都空闲后, 这时就可以调用 `free()`方法将共享内存释放, `free()`方法会调用 `POSIX.munmap()`方法删除共享内存文件的内存映射。

```
public void free() {
    try {
        POSIX.munmap(baseAddress, mmapLength);
    } catch (IOException e) {
        LOG.warn(this + ": failed to munmap", e);
    }
    LOG.trace(this + ": freed");
}
```

2. DfsClientShm 类

`DfsClientShm` 类用于在 `DFSClient` 侧抽象共享内存, 这个类是 `ShortCircuitShm` 的子类。要注意这个类实现了 `DomainSocketWatcher.Handler` 接口, 当 `DFSClient` 和 `Datanode` 进程通信的 `domainSocket` 对象关闭时, 这个 `DfsClientShm` 会被标识为 `stale` 状态。`stale` 状态的 `DfsClientShm` 中保存的所有 `Slot` 对象会被设置为无效, 同时 `Slot` 对象对应数据块的 `ShortCircuitReplica` 对象会被同步设置为 `stale` 状态, 也就不能再接收新的读请求以及缓存请求了, 但是当前正在进行的读操作不会受到影响。当一个 `Slot` 对象对应的 `ShortCircuitReplica` 上所有的读取操作完成后, `ShortCircuitCache` 就会关闭这个 `ShortCircuitReplica`, 并且释放对应的槽位。如果共享内存段中所有的槽位都已经释放, 就可以释放这段共享内存了。之所以这样设计, 是因为如果共享内存匹配的 `domainSocket` 关闭后, `Datanode` 侧的共享内存状态和 `DFSClient` 侧的共享内存不能确保是同步更新的, 所以需要引入 `DomainSocketWatcher.Handler` 接口处理 `domainSocket` 关闭的情况。

我们看一下处理上述逻辑的 `handle()` 方法的实现。

```
public boolean handle(DomainSocket sock) {
    manager.unregisterShm(getShmId());
    synchronized (this) {
        Preconditions.checkNotNull(!stale);
        stale = true; // 将共享内存段设置为 stale 状态
        boolean hadSlots = false;
        // 将共享内存中的所有 Slot 都设置为无效
        for (Iterator<Slot> iter = slotIterator(); iter.hasNext(); ) {
            Slot slot = iter.next();
            slot.makeInvalid();
            hadSlots = true;
        }
        if (!hadSlots) {
            // 如果共享内存中的所有槽位都被释放了，则调用 free() 方法释放共享内存
            free();
        }
    }
    return true;
}
```

3. DfsClientShmManager 类

`DfsClientShmManager` 管理着 HDFS 客户端侧的所有共享内存，如图 5-23 所示，一个 `DFSCClient` 可能需要维护与多个 `Datanode` 之间的共享内存，所以 `DfsClientShmManager` 定义了内部类 `EndpointShmManager` 来管理与一个 `Datanode` 之间的所有共享内存，每段共享内存又由一个 `DfsClientShm` 对象管理。

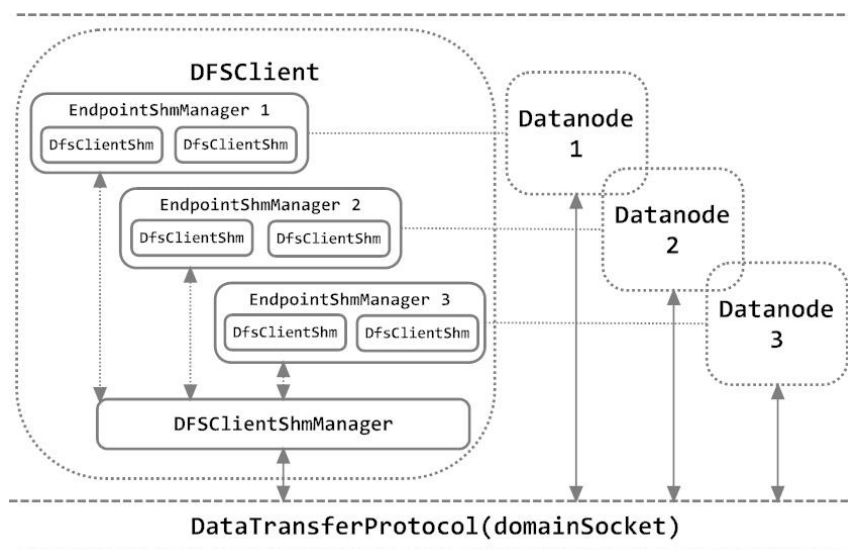


图 5-23 DfsClientShmManager 结构图

DfsClientShmManager 类结构如图 5-24 所示，它定义了 datanodes 字段保存所有的 EndpointShmManager 对象，同时对外提供 allocSlot()和 freeSlot()两个方法分别用于从指定 Datanode 的共享内存中分配和释放一个槽位。allocSlot()和 freeSlot()方法首先都会从 datanodes 字段中提取出该 Datanode 对应的 EndpointShmManager 对象，然后调用 EndpointShmManager 的 allocSlot()和 freeSlot()方法执行获取和释放槽位的操作。

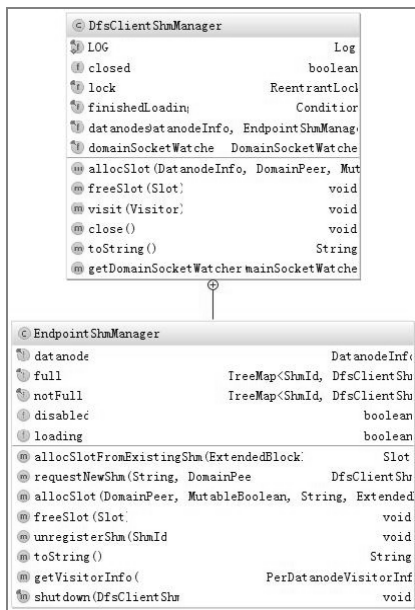


图 5-24 DfsClientShmManager 类结构

DFSClient 和一个 Datanode 之间可能有多段共享内存，其中有些共享内存可能已经没有可用槽位了，所以 EndpointShmManager 定义了两个队列：notFull 队列存储还有槽位的共享内存；full 队列存储没有槽位的共享内存。如图 5-24 所示，EndpointShmManager 类还定义了如下几个方法。

- allocSlot()方法：用来在共享内存中申请一个槽位。
- freeSlot()方法：用于释放一个槽位。allocSlot()和 freeSlot()这两个方法都是在 ShortCircuitCache 创建或者关闭一个 ShortCircuitReplica 时调用的。
- requestNewShm()方法：用于与 Datanode 协商创建一段新的共享内存。

下面我们就依次看一下这三个方法的实现。

(1) allocSlot()

allocSlot()方法实现了分配槽位的操作。这个方法首先从 notFull 队列中的共享内存中申请一个槽位。如果当前 EndpointShmManager 管理的所有共享内存中已经没有槽位了，则调用 requestNewShm()方法通过 DataTransferProtocol 与 Datanode 申请一段新的共享内存。申请完

成后，构造 `DfsClientShm` 对象管理这段共享内存，同时触发 `DfsClientShm` 监控底层的 `domainSocket`，并将这个新构造的 `DfsClientShm` 对象放入 `notFull` 队列中。`allocSlot()`方法的代码如下：

```
Slot allocSlot(DomainPeer peer, MutableBoolean usedPeer,
    String clientName, ExtendedBlockId blockId) throws IOException {
    while (true) {
        if (closed) {
            // DfsClientShmManager 已经关闭了，返回 null
            return null;
        }
        if (disabled) {
            // 如果当前 EndpointShmManager 对应的 Datanode 不支持短路读取，则返回 null
            return null;
        }
        // 如果已经申请了共享内存，则从该共享内存中申请槽位
        Slot slot = allocSlotFromExistingShm(blockId);
        if (slot != null) {
            return slot;
        }
        // 没有可用的槽位，但是已经有线程正在申请新的共享内存，则等待
        if (loading) {
            finishedLoading.awaitUninterruptibly();
        } else {
            // 当前线程申请新的共享内存
            loading = true;
            lock.unlock();
            DfsClientShm shm;
            try {
                // 通过 DataTransferProtocol 申请新的共享内存
                shm = requestNewShm(clientName, peer);
                if (shm == null) continue;
                // 将已经申请的共享内存加入 domainSocketWather，监控状态
                domainSocketWatcher.add(peer.getDomainSocket(), shm);
                usedPeer.setValue(true);
            } finally {
                lock.lock();
                loading = false;
                // 通知其他所有等待的线程
                finishedLoading.signalAll();
            }
            if (shm.isStale()) {
                // 如果刚刚申请的共享内存已经过期了，那么继续循环
            } else {
                // 将刚刚申请的共享内存加入 notFull 队列中
                notFull.put(shm.getShmId(), shm);
            }
        }
    }
}
```

(2) freeSlot()

`freeSlot()`方法用于释放一个 Slot，与 `allocSlot()`方法的实现很类似。它首先会从共享内存中释放这个槽位，然后根据共享内存的状态执行对应的操作，具体逻辑请参考代码及注释。

```
void freeSlot(Slot slot) {
    DfsClientShm shm = (DfsClientShm)slot.getShm();
    // 从共享内存中释放槽位
    shm.unregisterSlot(slot.getSlotIdx());
    if (shm.isStale()) {
        // 如果共享内存已经被标记为过期，并且共享内存中的所有槽位都被释放，则将共享内存释放
        if (shm.isEmpty()) {
            shm.free();
        }
    } else {
        ShmId shmId = shm.getShmId();
        // 由于共享内存释放了一个槽位，那么这个共享内存就不再是 full 状态了，从 full 队列中移除
        full.remove(shmId);
        // 如果共享内存中的所有槽位都空闲，则调用 shutdown() 方法将共享内存相关的 domainSocket 关闭。
        // 这时 DomainSocketWatcher 会调用 DfsClientShm.handle() 方法清理共享内存段
        if (shm.isEmpty()) {
            notFull.remove(shmId);
            shutdown(shm);
        } else {
            // 否则，就直接将共享内存放入 notFull 队列中
            notFull.put(shmId, shm);
        }
    }
}
```

(3) requestNewShm()

`requestNewShm()`方法用于在共享内存中没有可用槽位时，与 `Datanode` 协商构造一段新的共享内存。`requestNewShm()`方法会调用 `DataTransferProtocol.requestShortCircuitShm()`方法向 `Datanode` 申请一段共享内存，`Datanode` 会通过 `domainSocket` 将共享内存文件的文件描述符发回 `EndpointShmManager`。`EndpointShmManager` 会打开共享内存文件并执行内存映射操作，然后 `EndpointShmManager` 会构造 `DfsClientShm` 对象并返回。

```
private DfsClientShm requestNewShm(String clientName, DomainPeer peer)
    throws IOException {
    final DataOutputStream out =
        new DataOutputStream(
            new BufferedOutputStream(peer.getOutputStream()));
    // 调用 DataTransferProtocol.requestShortCircuitShm() 方法向 Datanode 申请一段共享内存
    new Sender(out).requestShortCircuitShm(clientName);
    // 接收响应消息
    ShortCircuitShmResponseProto resp =
        ShortCircuitShmResponseProto.parseFrom(
            PBHelper.vintPrefixed(peer.getInputStream()));
```



```

String error = resp.hasError() ? resp.getError() : "(unknown)";
switch (resp.getStatus()) {
case SUCCESS: // 如果成功
    DomainSocket sock = peer.getDomainSocket();
    byte buf[] = new byte[1];
    FileInputStream fis[] = new FileInputStream[1];
    // 从 domainSocket 接收共享内存文件的文件描述符
    if (sock.recvFileInputStreams(fis, buf, 0, buf.length) < 0) {
        throw new EOFException("got EOF while trying to transfer the " +
            "file descriptor for the shared memory segment.");
    }
    if (fis[0] == null) {
        throw new IOException("the datanode " + datanode + " failed to " +
            "pass a file descriptor for the shared memory segment.");
    }
    try {
        // 构造 DfsClientShm 对象
        DfsClientShm shm =
            new DfsClientShm(PBHelper.convert(resp.getId()),
                fis[0], this, peer);
        return shm;
    } finally {
        IOUtils.cleanup(LOG, fis[0]);
    }
case ERROR_UNSUPPORTED:
    disabled = true;
    return null;
default:
    return null;
}
}

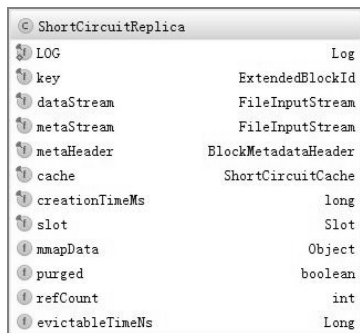
```

4. ShortCircuitReplica 类

ShortCircuitReplica 类封装了一个短路读数据块副本的所有信息，只有获取了 ShortCircuitReplica 对象，才能构造 BlockReaderLocal 对象完成短路读操作。

我们首先看一下 ShortCircuitReplica 类定义的字段，如图 5-25 所示，这个类中的每一个字段都很重要。

- **key**: 对应 BlockId，唯一标识这个 ShortCircuitReplica 对象。
- **dataStream**: 副本的数据块文件输入流。
- **metaStream**: 副本的校验文件输入流。
- **metaHeader**: 副本校验文件头。



Field Name	Type
LOG	Log
key	ExtendedBlockId
dataStream	FileInputStream
metaStream	FileInputStream
metaHeader	BlockMetadataHeader
cache	ShortCircuitCache
creationTimeMs	long
slot	Slot
mmapData	Object
purged	boolean
refCount	int
evictableTimeNs	Long

图 5-25 ShortCircuitReplica 类定义的字段

- **cache**: 管理这个 ShortCircuitReplica 对象的 ShortCircuitCache 对象。
- **slot**: 这个 ShortCircuitReplica 对象对应的共享内存中的槽位。
- **mmapData**: 当前 ShortCircuitReplica 对象在内存中的映射数据。
- **purged**: 当前 ShortCircuitReplica 对象是否在 cache 中被删除。
- **refCount**: 非常重要的一个字段，标识当前 ShortCircuitReplica 对象被引用的次数。ShortCircuitReplica 只可能被 ShortCircuitCache、BlockReaderLocal 以及 ClientMmap 对象引用。当我们构建一个 ShortCircuitReplica 对象时，这个值为 2，因为在创建时，ShortCircuitReplica 对象就已经被 ShortCircuitCache 以及请求类引用了。只有 refCount==0 时，ShortCircuitReplica 对象才可以被关闭。
- **evictableTimeNs**: ShortCircuitReplica 对象被加入 ShortCircuitCache 的 evictable 队列的时间。如果不在 evictable 队列中，则为 null。

ShortCircuitReplica 类还定义了几个比较重要的方法，如 unref() 方法、loadMmapInternal() 方法以及 addNoChecksumAnchor() 方法。unref() 用于在不需要对 ShortCircuitReplica 对象的引用时，解除对 ShortCircuitReplica 对象的引用。这个方法调用了 ShortCircuitCache 的同名方法进行解关联操作（我们在下一小节中介绍 ShortCircuitCache.unref() 方法）。unref() 方法的代码如下：

```
public void unref() {
    cache.unref(this);
}
```

loadMmapInternal() 方法用于将 ShortCircuitReplica 对应的数据块文件映射到内存中，这个方法是在客户端对副本进行零拷贝读取时调用的，这个方法的逆方法是 munmap()。loadMmapInternal() 方法的代码如下：

```
MappedByteBuffer loadMmapInternal() {
    try {
        FileChannel channel = dataStream.getChannel();
        // 调用 FileChannel.map() 方法将数据块文件映射到内存中
        MappedByteBuffer mmap = channel.map(MapMode.READ_ONLY, 0,
            Math.min(Integer.MAX_VALUE, channel.size()));
        return mmap;
    }
```

```

    } catch (IOException e) {
        return null;
    } catch (RuntimeException e) {
        return null;
    }
}

void munmap() {
    MappedByteBuffer mmap = (MappedByteBuffer)mmapData;
    // 调用 munmap() 系统调用移除映射
    NativeIO.POSIX.munmap(mmap);
    mmapData = null;
}

```

`addNoChecksumAnchor()`方法用于在共享内存的槽位中添加一个免校验（no-checksum）的锚（anchor）。只有当 `Datanode` 通过 `mlock` 操作将数据块副本缓存到内存后，也就是副本对应的 `Slot` 是可锚状态时，才可以添加这个锚。`addNoChecksumAnchor()`方法直接调用了 `Slot.addAnchor()`方法执行这个操作。方法的代码如下：

```

public boolean addNoChecksumAnchor() {
    if (slot == null) {
        return false;
    }
    boolean result = slot.addAnchor();
    return result;
}

```

5. ShortCircuitCache 类

`ShortCircuitCache` 是 `DFSClient` 短路读取操作中最重要类，负责对 `DFSClient` 的所有 `ShortCircuitReplica` 对象进行缓存以及生命周期管理等操作，`ShortCircuitReplica` 在 `ShortCircuitCache` 中的状态如图 5-26 所示。

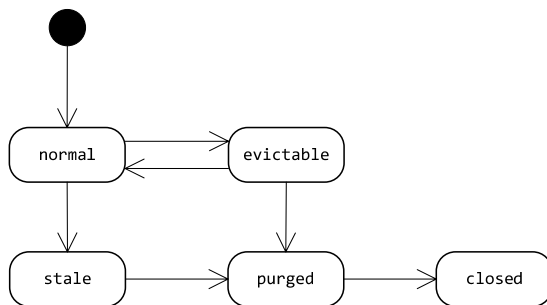


图 5-26 `ShortCircuitReplica` 对象状态图

`ShortCircuitReplica` 在 `ShortCircuitCache` 中会处于如下几种状态。

- **normal**: `ShortCircuitReplica` 最正常状态，可以随时通过 `fetch()`方法从缓存中取出。

- **evictable**: 可移除状态, 当 `DFSClient` 中只有 `ShortCircuitCache` 类引用了该副本, 没有其他类引用时, 副本处于可移除状态。`ShortCircuitCache` 会定期清理缓存中的可移除副本。
- **purged**: 删除状态, 副本已经从缓存中删除了, `DFSClient` 不能通过缓存获取该副本。处于删除状态的副本不一定被关闭了, 有可能还有引用这个副本的操作, 当引用数等于 0 时, 就可以将副本关闭了。
- **closed**: 关闭状态, 副本对应的 `Slot` 已经从共享内存中释放了。
- **stale**: 过期状态, 有可能是副本对应的 `Slot` 无效, 也有可能是副本对应的输入流异常, 还有可能是副本在缓存中存在的时间超过有效时间。

下面我们看一下可能发生状态转移的情况。

- **起始 -> normal**: 用户调用 `fetchAndCreat()` 方法创建一个副本时, 这时副本处于缓存中, 则副本为正常状态。
- **normal -> stale**:
 - `fetch()` 副本时发现副本对应的 `Slot` 无效, 或者是副本在缓存中存在的时间超过有效时间, 则将副本状态设置为 `stale` 状态。
 - `unref()` 操作时发现副本对应的输入流异常, 则将副本状态设置为 `stale` 状态。
- **stale -> purged**: 一旦发现 `stale` 状态的副本, 就马上调用 `purge()` 方法将副本从缓存中删除, 将副本状态设置为 `purged` 状态。
- **normal -> evictable**: `unref()` 操作时, 发现 `refCount==1`, 也就是只有 `ShortCircuitCache` 引用了当前副本, 这时就将副本放入 `evictable` 队列中, 将副本状态设置为 `evictable` 状态。
- **evictable -> normal**: `ref()` 操作时, `refCount` 增加, 则将副本移出 `evictable` 队列, 将副本状态恢复为正常状态。
- **evictable -> purged**:
 - `CacheCleaner` 发现 `evictable` 队列中有超时副本, 则将超时副本删除。
 - `trimEnviacationMap()`: 当我们向 `evictable` 队列中添加元素时, 会触发 `trimEnviacationMap()` 方法判断是否超过缓存数量。如果超过, 则从缓存队列中直接删除副本。
- **purged -> closed**: 当 `refCount == 0` 时, 也就是完全没有对象引用这个副本时, 调用副本的 `close()` 方法。

下面我们看一下这些涉及状态转移方法的实现。

(1) `fetchOrCreate()`

`fetchOrCreate()` 方法用于从 `ShortCircuitCache` 缓存中获取一个 `ShortCircuitReplicaInfo` 对象 (封装了 `ShortCircuitReplica` 对象), 或者在缓存没有保存这个对象时创建一个新的 `ShortCircuitReplicaInfo` 对象。

需要特别注意的是, 可能同时有多个线程并发地调用这个方法来获取同一个 `ShortCircuit`

ReplicaInfo 对象, fetchOrCreate()会允许第一个申请创建 ShortCircuitReplicaInfo 对象的线程调用 create()方法执行创建操作, 同时 fetchOrCreate()方法会在缓存 ShortCircuitReplicaInfo 对象的 ShortCircuitCache.replicaInfoMap 字段中放入一个 Waitable 对象。当其他的线程调用 fetchOrCreate()方法申请获取同一个 ShortCircuitReplicaInfo 对象时, 会得到这个 Waitable 对象并在这个 Waitable 对象上等待。当创建线程成功地创建 ShortCircuitReplicaInfo 对象后, 会唤醒该线程并通过 Waitable 对象将 ShortCircuitReplicaInfo 传递给该线程。通过这种同步机制, 多个线程之间可以并发地访问缓存, 值得我们学习。fetchOrCreate()方法的代码如下:

```
public ShortCircuitReplicaInfo fetchOrCreate(ExtendedBlockId key,
    ShortCircuitReplicaCreator creator) {
    Waitable<ShortCircuitReplicaInfo> newWaitable = null;
    lock.lock();
    try {
        ShortCircuitReplicaInfo info = null;
        do {
            if (closed) {
                return null;
            }
            // 这里从 replicaInfoMap 中获取 Waitable 对象
            Waitable<ShortCircuitReplicaInfo> waitable = replicaInfoMap.get(key);
            if (waitable != null) { // 已经有线程构造 ShortCircuitReplicaInfo 对象的情况
                try {
                    // 调用 fetch() 方法等待获取 ShortCircuitReplicaInfo 对象
                    info = fetch(key, waitable);
                } catch (RetriableException e) {
                    continue;
                }
            }
        } while (false);
        if (info != null) return info;
        // 第一个尝试创建 ShortCircuitReplicaInfo 对象的线程, 先构造 Waitable 并放入 replicaInfoMap
        newWaitable = new Waitable<ShortCircuitReplicaInfo>(lock.newCondition());
        replicaInfoMap.put(key, newWaitable);
    } finally {
        lock.unlock();
    }
    // 调用 create() 方法创建 ShortCircuitReplicaInfo 对象, 并返回
    return create(key, creator, newWaitable);
}
```

可见 fetchOrCreate()方法将任务委派给了 fetch()以及 create()方法, 我们看一下这两个方法的实现。fetch()方法会在 Waitable 对象上调用 await()方法, 等待另一个线程完成 ShortCircuitReplica 对象的创建操作。这里需要注意的是, 对于 purged 以及 stale 状态的 ShortCircuitReplica 对象, fetch()方法会抛出 RetriableException 异常。同时由于 fetch()方法的调用类会获取 ShortCircuitReplica 对象的引用, 所以还需要调用 ref()操作增加副本的引用次数。

fetch()方法的代码如下:

```
private ShortCircuitReplicaInfo fetch(ExtendedBlockId key,
    Waitable<ShortCircuitReplicaInfo> waitable) throws RetriableException {

    ShortCircuitReplicaInfo info;
    try {
        // 在 Waitable 对象上等待, 等待另一个线程构造 ShortCircuitReplicaInfo 对象
        info = waitable.await();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new RetriableException("interrupted");
    }
    // ... 安全相关
    ShortCircuitReplica replica = info.getReplica();
    if (replica == null) {
        LOG.warn(this + ": failed to get " + key);
        return info;
    }
    if (replica.purged) {
        // 如果当前取得的副本已经从缓存中清除了, 则抛出异常, 在 fetchAndCreate() 方法中调用 create()
        // 创建一个新的副本
        throw new RetriableException("Ignoring purged replica " +
            replica + ". Retrying.");
    }
    // 如果当前副本已经过期了, 则从缓存中清除该副本, 抛出异常, 在 fetchAndCreate() 方法中调用 create()
    // 创建一个新的副本
    if (replica.isStale()) {
        purge(replica);
        throw new RetriableException("ignoring stale replica " + replica);
    }
    // 在副本上添加引用计数, 因为是获取一个已经创建的副本, 所以需要添加新的引用计数
    ref(replica);
    return info;
}
```

了解了 fetch()方法, 现在我们看一下 create()方法的实现。create()方法的逻辑在图 5-19 中已经给出了, 它首先会调用 BlockReaderFacotry.createShortCircuitReplicaInfo() 方法创建 ShortCircuitReplicaInfo 对象, 创建成功之后会将新创建的 ShortCircuitReplicaInfo 添加到 Waitable 对象中, 并且唤醒所有在 Waitable 上等待的线程。

```
private ShortCircuitReplicaInfo create(ExtendedBlockId key,
    ShortCircuitReplicaCreator creator,
    Waitable<ShortCircuitReplicaInfo> newWaitable) {
    ShortCircuitReplicaInfo info = null;
    try {
        // 这里调用了 BlockReaderFacotry.createShortCircuitReplicaInfo() 方法创建
        // ShortCircuitReplicaInfo 对象
        info = creator.createShortCircuitReplicaInfo();
    }
```

```

    } catch (RuntimeException e) {
        LOG.warn(this + ": failed to load " + key, e);
    }
    // 否则, 调用构造方法构造这个 ShortCircuitReplicaInfo 对象
    if (info == null) info = new ShortCircuitReplicaInfo();
    lock.lock();
    try {
        if (info.getReplica() != null) {
            // 创建成功, 确保启动了 CacheCleaner
        } else {
            // 创建失败, 从 replicaInfoMap 中删除 Waitable
            Waitable<ShortCircuitReplicaInfo> waitableInMap = replicaInfoMap.get(key);
            if (waitableInMap == newWaitable) replicaInfoMap.remove(key);
        }
        // 将 ShortCircuitReplicaInfo 添加到 Waitable 中, 并且唤醒所有在 Waitable 上等待的线程
        newWaitable.provide(info);
    } finally {
        lock.unlock();
    }
    return info;
}

```

(2) purge()

上述两个方法都用于在缓存中添加 `ShortCircuitReplica` 对象, 下面我们介绍从缓存中删除 `ShortCircuitReplica` 的 `purge()` 方法。要注意 `purge()` 方法只是将副本从缓存中移除, 并没有关闭 `ShortCircuitReplica` 对象, 也就是没有释放共享内存中的槽位。`Purge()` 方法首先会设置副本的状态为 `purged`, 然后将 `ShortCircuitReplica` 对象从 `replicaInfoMap` 以及 `evictable` 队列中删除。最后由于已经从缓存中删除了 `ShortCircuitReplica` 对象, 所以调用 `unref()` 方法降低副本的引用计数。

```

private void purge(ShortCircuitReplica replica) {
    boolean removedFromInfoMap = false;
    String evictionMapName = null;
    Preconditions.checkArgument(!replica.purged);
    // replica 已经删除
    replica.purged = true;
    Waitable<ShortCircuitReplicaInfo> val = replicaInfoMap.get(replica.key);
    // 从 replicaInfoMap 中删除数据
    if (val != null) {
        ShortCircuitReplicaInfo info = val.getVal();
        if ((info != null) && (info.getReplica() == replica)) {
            replicaInfoMap.remove(replica.key);
            removedFromInfoMap = true;
        }
    }
    // 从 evictable 队列中删除副本数据
    Long evictableTimeNs = replica.getEvictableTimeNs();
}

```

Hadoop 2.X HDFS 源码剖析

```
if (evictableTimeNs != null) {
    evictionMapName = removeEvictable(replica);
}
// 由于在 replica 构造时就考虑了缓存的引用，所以从缓存中删除时，要 unref() 这个 replica
unref(replica);
}
```

这里我们看一下 unref() 操作。unref() 操作只能在 ShortCircuitReplica.unref() 以及 purge() 方法中被调用，也就是在使用类不再引用 ShortCircuitReplica 对象以及缓存中不存在 ShortCircuitReplica 对象时才会被调用。如果当前副本已经过期或者不可用，unref() 方法会直接将当前副本从缓存中删除。如果当前副本没有被引用，也就是没有类使用这个副本进行读取操作，同时副本已经从缓存中删除了，这种情况就需要将副本关闭。关闭副本操作会释放副本对应的 Slot 对象，并且释放副本数据块文件的内存映射。如果副本没有被引用，但在缓存中还存在，则将副本移入 evictable 队列，向 evictable 队列中添加了元素后，还需要调用 trimEvictionMaps() 方法将超过缓存队列大小的副本移出缓存。unref() 方法的代码如下：

```
void unref(ShortCircuitReplica replica) {
    lock.lock();
    try {
        // 如果当前副本已经过期或者不可用，则直接将当前副本从缓存中删除
        if (!replica.purged) {
            String purgeReason = null;
            if (!replica.getDataStream().getChannel().isOpen()) {
                purgeReason = "purging replica because its data channel is closed.";
            } else if (!replica.getMetaStream().getChannel().isOpen()) {
                purgeReason = "purging replica because its meta channel is closed.";
            } else if (replica.isStale()) {
                purgeReason = "purging replica because it is stale.";
            }
            if (purgeReason != null) {
                purge(replica);
            }
        }
        String addedString = "";
        boolean shouldTrimEvictionMaps = false;
        // 更新 refCount
        int newRefCount = --replica.refCount;
        // 副本没有被引用，也就是没有类使用这个副本进行读取操作，同时副本也从缓存中删除了，这种情况就将副本关闭，释放 Slot，释放副本数据块文件的内存映射
        if (newRefCount == 0) {
            Preconditions.checkArgument(replica.purged,
                "Replica " + replica + " reached a refCount of 0 without " +
                "being purged");
            replica.close();
        } else if (newRefCount == 1) {
            // 副本没有被引用，也就是没有类使用这个副本进行读取操作，但在缓存中还存在，则将副本移入 evictable 队列
        }
    }
}
```



```

Preconditions.checkState(null == replica.getEvictableTimeNs(),
    "Replica " + replica + " had a refCount higher than 1, " +
    "but was still evictable (evictableTimeNs = " +
    replica.getEvictableTimeNs() + ")");
if (!replica.purged) {
    // 加入对应的 evictable 队列
    if (replica.hasMmap()) {
        insertEvictable(System.nanoTime(), replica, evictableMmapped);
        addedString = "added to evictableMmapped, ";
    } else {
        insertEvictable(System.nanoTime(), replica, evictable);
        addedString = "added to evictable, ";
    }
    shouldTrimEvictionMaps = true;
}
} else {
    Preconditions.checkArgument(replica.refCount >= 0,
        "replica's refCount went negative (refCount = " +
        replica.refCount + " for " + replica + ")");
}
// 调整 evictable 队列的大小, 大于缓存队列大小的副本直接删除
if (shouldTrimEvictionMaps) {
    trimEvictionMaps();
}
} finally {
    lock.unlock();
}
}

```

(3) evictable 队列操作

通过上述方法我们知道, 当缓存中的副本没有被对象引用时, 就会被放入 `eviCtable` 队列中。由于有些副本执行了零拷贝读操作, 也就是副本的数据块文件被映射到了内存中, 这种已经进行了 `mmap` 的副本, 我们会将它放入 `evictableMmapped` 队列中保存, 而没有进行 `mmap` 操作的副本则放入另一个队列 `evictable` 中。当需要从队列中删除元素时, 优先从 `evictable` 队列中删除, 因为 `mmap` 操作更加耗时。

向 `evictable` 以及 `evictableMmapped` 中添加元素, 我们已经介绍过了, 当副本的 `refCount==1` 时, 就会被添加到 `evictable` 队列中。那什么时候删除呢? 当 `ShortCircuitCache` 创建了一个副本时, 就会尝试启动 `CacheCleaner` 线程。`CacheCleaner` 线程会尝试调用 `demoteOldEvictableMmapped()` 方法将 `evictableMmapped` 中的超时副本 (超过 `maxEvictableMmappedLifespanMs` 时间) 放入 `evictable` 队列中, 然后将 `evictable` 队列中的超时副本 (超过 `maxNonMmappedEvictableLifespanMs` 时间) 从缓存中删除。`CacheCleaner` 代码如下:

```

public void run() {
    ShortCircuitCache.this.lock.lock();
    try {

```

```
if (ShortCircuitCache.this.closed) return;
long curMs = Time.monotonicNow();
// 将 evictableMmapped 队列中的元素放入 evictable 队列中
int numDemoted = demoteOldEvictableMmapped(curMs);
int numPurged = 0;
Long evictionTimeNs = Long.valueOf(0);
while (true) {
    // evictable 是 treeMap, 按序拿出第一个副本
    Entry<Long, ShortCircuitReplica> entry =
        evictable.ceilingEntry(evictionTimeNs);
    if (entry == null) break;
    evictionTimeNs = entry.getKey();
    long evictionTimeMs =
        TimeUnit.MILLISECONDS.convert(evictionTimeNs, TimeUnit.NANOSECONDS);
    // 大于 maxNonMmappedEvictableLifespanMs 时间的, 则直接删除
    if (evictionTimeMs + maxNonMmappedEvictableLifespanMs >= curMs) break;
    ShortCircuitReplica replica = entry.getValue();
    // 调用 purge() 方法删除这个副本
    purge(replica);
    numPurged++;
}
} finally {
    ShortCircuitCache.this.lock.unlock();
}
}
```

(4) 关闭 ShortCircuitReplica 对象

当一个 ShortCircuitReplica 对象的 refCount 等于 0 时, 也就是没有任何对象引用这个 ShortCircuitReplica 对象时, 就可以调用 ShortCircuitReplica.close() 方法关闭这个 ShortCircuitReplica 对象了。ShortCircuitReplica.close()方法首先会调用 munmap 系统调用删除数据块文件在内存中的映射数据, 然后调用 ShortCircuitCache.scheduleSlotReleaser()方法释放副本对应的槽位。

```
void close() {
    String suffix = "";
    refCount = -1;
    if (hasMmap()) {
        munmap();
        suffix += " munmapped.";
    }
    IOUtils.cleanup(LOG, dataStream, metaStream);
    if (slot != null) {
        cache.scheduleSlotReleaser(slot);
        suffix += " scheduling " + slot + " for later release.";
    }
}
```

下面我们看一下 scheduleSlotReleaser()方法的实现。它首先调用 DataTransferProtocol.

`releaseShortCircuitFds()`方法通知 `Datanode` 释放 `Datanode` 侧共享内存中的槽位，然后调用 `shmManager.freeSlot()`方法释放 `Client` 侧共享内存中的槽位。如果在 `RPC` 过程中失败，则直接关闭这个共享内存段，因为 `Client` 与 `Datanode` 之间的通信已经出现了异常。

```
public void scheduleSlotReleaser(Slot slot) {
    Preconditions.checkNotNull(shmManager);
    releaserExecutor.execute(new SlotReleaser(slot));
}

public void run() {
    final DfsClientShm shm = (DfsClientShm)slot.getShm();
    final DomainSocket shmSock = shm.getPeer().getDomainSocket();
    DomainSocket sock = null;
    DataOutputStream out = null;
    final String path = shmSock.getPath();
    boolean success = false;
    try {
        sock = DomainSocket.connect(path);
        out = new DataOutputStream(
            new BufferedOutputStream(sock.getOutputStream()));
        new Sender(out).releaseShortCircuitFds(slot.getSlotId());
        DataInputStream in = new DataInputStream(sock.getInputStream());
        ReleaseShortCircuitAccessResponseProto resp =
            ReleaseShortCircuitAccessResponseProto.parseFrom(
                PBHelper.vintPrefixed(in));
        if (resp.getStatus() != Status.SUCCESS) {
            String error = resp.hasError() ? resp.getError() : "(unknown)";
            throw new IOException(resp.getStatus().toString() + ": " + error);
        }
        success = true;
    } catch (IOException e) {

    } finally {
        if (success) {
            shmManager.freeSlot(slot);
        } else {
            shm.getEndpointShmManager().shutdown(shm);
        }
        IOUtils.cleanup(LOG, sock, out);
    }
}
```

5.3.4 Datanode 短路读操作流程

了解了 `Client` 侧发起短路读操作的流程后，我们在这一节学习 `Datanode` 是如何响应 `Client` 的短路读请求的。`Datanode` 中与短路读操作相关的类主要有两个。

- **RegisteredShm**: **ShortCircuitShm** 的子类, **ShortCircuitRegistry** 的内部类, 用来描述 **Datanode** 侧的一段共享内存。
- **ShortCircuitRegistry**: 管理 **Datanode** 侧的所有共享内存。客户端使用 **DataTransferProtocol** 申请新的共享内存段以及释放已有的共享内存段时, 是由 **ShortCircuitRegistry** 类来执行对应的操作的。

1. RegisteredShm 类

DfsClientShm 类用于描述 **DFSClient** 侧的共享内存, 与 **DfsClientShm** 一样, **RegisteredShm** 也是 **ShortCircuitRegistry** 的子类, 同样实现了 **DomainSocketWatcher.Handler** 接口。

RegisteredShm 类的定义如下所示, 当 **Client** 与 **Datanode** 之间通信的 **domainSocket** 出现异常时, 会调用 **RegisteredShm.handle()** 方法处理这个异常。**RegisteredShm.handle()** 方法的逻辑很简单, 直接调用 **ShortCircuitRegistry.removeShm()** 方法从 **ShortCircuitRegistry** 删除这段共享内存即可。

```
private static class RegisteredShm extends ShortCircuitShm
    implements DomainSocketWatcher.Handler {
    private final ShortCircuitRegistry registry;

    RegisteredShm(ShmId shmId, FileInputStream stream,
        ShortCircuitRegistry registry) throws IOException {
        super(shmId, stream);
        this.registry = registry;
    }

    @Override
    public boolean handle(DomainSocket sock) {
        synchronized (registry) {
            synchronized (this) {
                // 调用 ShortCircuitRegistry.removeShm() 方法处理 domainSocket 关闭的情况
                registry.removeShm(this);
            }
        }
        return true;
    }
}
```

2. ShortCircuitRegistry 类

ShortCircuitRegistry 类用于管理 **Datanode** 侧的所有共享内存, 包括创建、释放共享内存等操作。

如图 5-27 所示, **ShortCircuitRegistry** 定义了 **segments** 字段保存所有的共享内存对象 **RegisteredShm**, 同时定义了 **slots** 字段保存所有的槽位对象 **Slot**。**ShortCircuitRegistry** 还定义了如下方法。

ShortCircuitRegistry	
LOG	Log
SHM_LENGTH	int
enabled	boolean
shmFactory	SharedFileDescriptorFactory
watcher	DomainSocketWatcher
segments	HashMap<ShmId, RegisteredShm>
slots	HashMultimap<ExtendedBlockId, Slot>
removeShm(ShortCircuitShm)	void
processBlockMlockEvent(ExtendedBlockId)	void
processBlockMunlockRequest(ExtendedBlockId)	boolean
processBlockInvalidation(ExtendedBlockId)	void
getClientNames(ExtendedBlockId)	String
createNewMemorySegment(String, DomainSocket) wShmInfo	
registerSlot(ExtendedBlockId, SlotId, boolean)	void
unregisterSlot(SlotId)	void
shutdown()	void

图 5-27 ShortCircuitRegistry 类结构

- **createNewMemorySegment():** 用于在 Datanode 侧构造一段共享内存，这个方法是在 `DataXceiver.requestShortCircuitShm()` 响应 `DataTransferProtocol.requestShortCircuitShm()` 请求时调用的（请参考 `DataTransferProtocol` 小节）。`createNewMemorySegment()` 方法会打开共享内存文件，并将该文件映射到内存中。然后创建 `RegisteredShm` 对象管理该共享内存，并将 `RegisteredShm` 对象加入 `ShortCircuitRegistry.segments` 字段中保存。
- **registerSlot():** 用于在 Datanode 侧的共享内存中添加一个 `Slot` 对象，这个方法是在 `DataXceiver.requestShortCircuitFds()` 响应 `DataTransferProtocol.requestShortCircuitFds()` 请求时调用的。`registerSlot()` 方法首先从 `ShortCircuitRegistry.segments` 字段中取出指定共享内存对应的 `RegisteredShm` 对象，然后调用 `RegisteredShm.registerSlot()` 方法构造并添加 `Slot` 对象，最后将该 `Slot` 对象加入 `ShortCircuitRegistry.slots` 字段中保存。
- **unregisterSlot():** 用于在 Datanode 侧的共享内存中删除一个 `Slot` 对象，这个方法是在 `DataXceiver.releaseShortCircuitFds()` 响应 `DataTransferProtocol.releaseShortCircuitFds()` 请求时调用的。`unregisterSlot()` 方法首先从 `ShortCircuitRegistry.segments` 字段中取出指定共享内存对应的 `RegisteredShm` 对象，然后调用 `RegisteredShm.unregisterSlot()` 释放 `Slot` 对象，最后将该 `Slot` 对象从 `ShortCircuitRegistry.slots` 字段中删除。
- **removeShm():** 用于从 `ShortCircuitRegistry` 类中删除一段共享内存，这个方法是在 Datanode 与 `DFSClient` 之间的 `DomainSocket` 出现异常时，由 `RegisteredShm.handle()` 方法调用的（请参考 `RegisteredShm` 类小节）。`removeShm()` 方法首先会停止 `ShortCircuitRegistry` 对这段共享内存的追踪，然后停止对共享内存中所有 `Slot` 对象的追踪，最后释放这段共享内存并关闭共享内存映射文件。
- **processBlockMlockEvent():** 用于在缓存数据块对应的 `Slot` 对象上添加可锚状态位，这个方法是在 Datanode 将一个数据块添加到缓存时，由 `CachingTask` 对象调用的（请参考第 4 章的 `FSDatasetImpl` 实现小节）。`processBlockMlockEvent()` 会从 `ShortCircuitRegistry.slots` 字段获取该数据块对应的 `Slot` 对象，然后调用 `Slot.makeAnchorable()` 方法添加一个可锚状态位，由于这个状态位是保存在共享内存文件中的，所以这个状

态位的信息会同步到 DFSCClient 端的 Slot 对象上。

- `processBlockMunlockRequest()`: 类似于 `processBlockMlockEvent()` 方法, 当 Datanode 将数据块从缓存中移除时, 会调用这个方法删除 Slot 对象的可锚状态位。

至此, 短路读取操作中涉及的类、流程以及方法就都介绍完了。下面我们开始学习客户端是如何进行写操作的。

5.4 文件写操作与输出流

介绍完客户端的读操作之后, 我们来学习文件写操作的实现。HDFS 是一个分布式文件系统, 不同于单机系统, HDFS 写文件操作非常复杂。整个写操作流程我们在 HDFS 概述章节中已经介绍过了, 本节就介绍 HDFS 写流程中与客户端相关的操作。

5.4.1 创建文件

客户端在执行文件写操作前, 首先需要调用 `DistributedFileSystem.create()` 创建一个空的 HDFS 文件, 并且获取这个 HDFS 文件的输出流 `HdfsDataOutputStream` 对象。成功获取到输出流对象后, 客户端就可以在输出流 `HdfsDataOutputStream` 对象上调用 `write()` 方法执行写操作了。

1. `DistributedFileSystem.create()`

用户代码创建一个新文件时, 会首先调用 `DistributedFileSystem.create()` 方法创建一个空文件, 然后通过 `create()` 方法返回的 `HdfsDataOutputStream` 输出流写文件。

如下代码所示, 这里 `DistributedFileSystem.create()` 方法直接调用了 `DFSCClient.create()` 方法, 并将返回的 `DFSOutputStream` 包装成一个 `HdfsDataOutputStream` 输出流。

```
public HdfsDataOutputStream create(final Path f,
    final FsPermission permission, final boolean overwrite,
    final int bufferSize, final short replication, final long blockSize,
    final Progressable progress, final InetSocketAddress[] favoredNodes)
    throws IOException {
    // ...
    return new FileSystemLinkResolver<HdfsDataOutputStream>() {
        @Override
        public HdfsDataOutputStream doCall(final Path p)
            throws IOException, UnresolvedLinkException {
            // 调用 DFSCClient.create() 创建 DFSOutputStream
            final DFSOutputStream out = dfs.create(getPathName(f), permission,
                overwrite ? EnumSet.of(CreateFlag.CREATE, CreateFlag.OVERWRITE)
                : EnumSet.of(CreateFlag.CREATE),
                true, replication, blockSize, progress, bufferSize, null,
                favoredNodes);
```

```

        // 构造 HdfsDataOutputStream, 包装 DFSOutputStream, 并返回
        return new HdfsDataOutputStream(out, statistics);
    }
    // ... next() 方法不再赘述
}

```

2. DFSClient.create()

上一节中我们介绍了 `DistributedFileSystem.create()` 会调用 `DFSClient.create()` 方法创建 `DFSOutputStream` 输出流, 并调用 `beginFileLease()` 方法获取文件的租约。这一节我们学习 `DFSClient.create()` 的代码实现。`DFSClient.create()` 的代码如下:

```

public DFSOutputStream create(String src,
                               FsPermission permission,
                               EnumSet<CreateFlag> flag,
                               boolean createParent,
                               short replication,
                               long blockSize,
                               Progressable progress,
                               int buffersize,
                               ChecksumOpt checksumOpt,
                               InetSocketAddress[] favoredNodes) throws IOException {
    // 检查客户端是否已经打开
    checkOpen();
    // ...
    // 调用 DFSOutputStream.newStreamForCreate() 创建 DFSOutputStream 对象
    final DFSOutputStream result = DFSOutputStream.newStreamForCreate(this,
        src, masked, flag, createParent, replication, blockSize, progress,
        buffersize, dfsClientConf.createChecksum(checksumOpt), favoredNodeStrs);
    beginFileLease(result.getFileId(), result);
    return result;
}

```

`DFSOutputStream` 中有两个静态的构造方法, 即 `newStreamForAppend()` 和 `newStreamForCreate()`。`newStreamForCreate()` 用于在客户端写一个新文件时构造 `DFSOutputStream` 输出流对象; `newStreamForAppend()` 则用于在客户端追加写一个已有文件时创建输出流对象。这两个方法调用了 `DFSOutputStream` 的不同构造方法, `newStreamForAppend()` 方法我们在追加写操作小节中介绍, 本节介绍 `newStreamForCreate()` 方法的实现。`DFSClient.create()` 方法调用流程如图 5-28 所示。

(1) DFSOutputStream.newStreamForCreate()

我们看一下 `newStreamForCreate()` 方法的实现。这个方法首先通过调用 `ClientProtocol.create()` 在 `Namenode` 的命名空间 (`Namespace`) 中创建一个新文件, 然后调用 `DFSOutputStream` 的构造方法创建输出流, 最后启动 `DFSOutputStream` 的 `streamer` 线程。`newStreamForCreate()` 方法的代码如下:

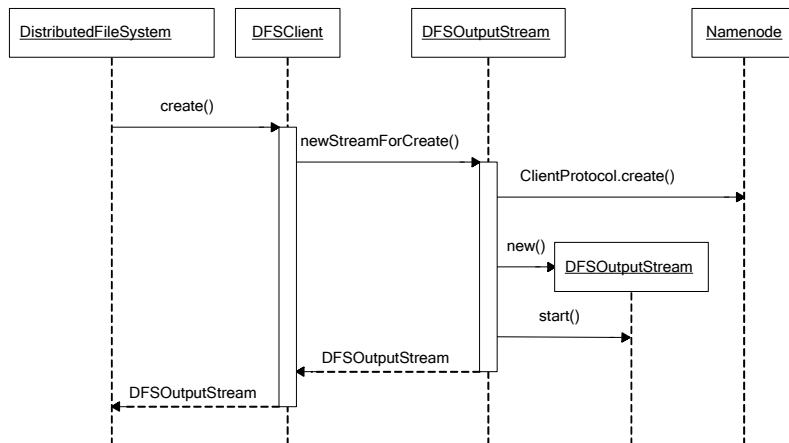


图 5-28 DFSClient.create()方法调用流程图

```

static DFSOutputStream newStreamForCreate(DFSClient dfsClient, String src,
    FsPermission masked, EnumSet<CreateFlag> flag, boolean createParent,
    short replication, long blockSize, Progressable progress, int buffersize,
    DataChecksum checksum, String[] favoredNodes) throws IOException {
    final HdfsFileStatus stat;
    try {
        // 调用 ClientProtocol.create() 方法，在命名空间中创建 HDFS 文件
        stat = dfsClient.namenode.create(src, masked, dfsClient.clientName,
            new EnumSetWritable<CreateFlag>(flag), createParent, replication,
            blockSize);
    } catch (RemoteException re) {
        throw re.unwrapRemoteException(AccessControlException.class,
            // ...
            SnapshotAccessControlException.class);
    }
    // 调用构造方法创建 DFSOutputStream 对象
    final DFSOutputStream out = new DFSOutputStream(dfsClient, src, stat,
        flag, progress, checksum, favoredNodes);
    // 启动 DFSOutputStream 的 streamer 线程
    out.start();
    return out;
}

```

下面我们就看一下 DFSOutputStream 构造方法的实现。

(2) DFSOutputStream 构造方法

DFSOutputStream 构造方法比较简单，它首先调用私有的构造方法初始化一些属性，并且对 shouldSyncBlock（是否在关闭时将数据块持久化到磁盘）属性赋值。然后调用 computePacketChunkSize() 方法确定数据包（packet）大小，同时确定一个数据包当中包含多少个校验块（chunk）。接下来 create() 方法会构造 streamer 线程，这个 streamer 线程是

DFSOutputStream 的内部类，它负责建立数据流管道（pipeline），并将数据包发送到数据流管道中的第一个 Datanode。create()方法最后设置了 favoredNodes 字段，确认客户端想要在某些 Datanode 上写入数据块。

```
private DFSOutputStream(DFSClient dfsClient, String src, HdfsFileStatus stat,
    EnumSet<CreateFlag> flag, Progressable progress,
    DataChecksum checksum, String[] favoredNodes) throws IOException {
    // 调用 DFSOutputStream 私有的构造方法初始化 DFSOutputStream 的字段
    this(dfsClient, src, progress, stat, checksum);
    // 初始化 shouldSyncBlock 字段
    this.shouldSyncBlock = flag.contains(CreateFlag.SYNC_BLOCK);
    // 调用 computePacketChunkSize() 方法计算 chunk 大小，以及 packet 中包含多少个 chunk
    computePacketChunkSize(dfsClient.getConf().writePacketSize,
        checksum.getBytesPerChecksum());
    // 构造 streamer 线程
    streamer = new DataStreamer();
    // 设置 favoredNodes 字段
    if (favoredNodes != null && favoredNodes.length != 0) {
        streamer.setFavoredNodes(favoredNodes);
    }
}
```

DFSOutputStream 的构造方法调用了 computePacketChunkSize()方法计算发送数据包大小，以及数据包中包含多少个校验块。在后续数据块的发送操作中，会使用这个方法中定义的数据包发送数据。computePacketChunkSize()方法的代码如下：

```
private void computePacketChunkSize(int psize, int csize) {
    // chunkSize 为完整校验块的大小，包括校验块数据和校验块数据对应的校验和
    final int chunkSize = csize + getChecksumSize();
    // 每个数据包中可以包含的校验块数量
    chunksPerPacket = Math.max(psize/chunkSize, 1);
    // 数据包的大小
    packetSize = chunkSize*chunksPerPacket;
}
```

computePacketChunkSize()中的 psize、csize 参数，以及产生的 chunksPerPacket、packetSize 等字段定义及参数值请参考表 5-2。

表 5-2 computePacketChunkSize()的参数及其含义

参数名称	参数值	参数含义
csize	512 字节	每个校验块的大小
psize	由 dfs.client-write-packet-size 配置项配置，默认为 64*1024 字节	每个数据包的最大字节数
chunkSize	512+4=516 字节	完整校验块的大小，一个完整的校验块包括校验块数据以及校验块对应的校验和数据
chunksPerPacket	127	每个数据包中可以包含的校验块数量
packetSize	516*127=65532	数据包的实际字节数

streamer 线程的构造方法及其使用我们在 DataStreamer 线程小节中介绍, 这里就不再详细介绍了。

5.4.2 写操作——DFSOutputStream 实现

当用户代码通过 DistributedFileSystem.create() 方法创建了一个新文件, 并获取了 DFSOutputStream 输出流对象之后, 就可以在输出流对象上调用 write() 方法写数据了。本节就介绍 DFSOutputStream 的实现, DFSOutputStream 类结构如图 5-29 所示。

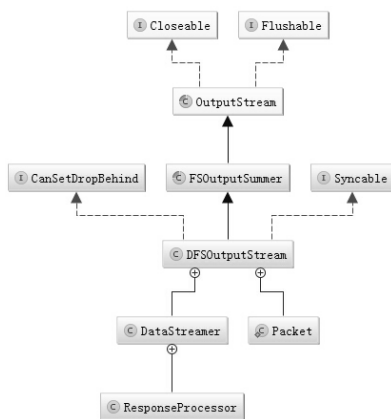


图 5-29 DFSOutputStream 类结构

DFSOutputStream 扩展自抽象类 FSOutputSummer, FSOutputSummer 在 OutputStream 的基础上提供了写数据并计算校验和的功能, DFSOutputStream.write() 方法的实现就继承自 FSOutputSummer 类。

DFSOutputStream 中使用 Packet 类来封装一个数据包。每个数据包中都包含若干个校验块, 以及校验块对应的校验和。一个完整的数据包结构如图 5-30 所示。首先是数据包包头, 记录了数据包的概要属性信息, 然后是校验和数据, 最后是校验块数据。Packet 类提供了 writeData() 以及 writeChecksum() 方法向数据块中写入校验块数据以及校验和。

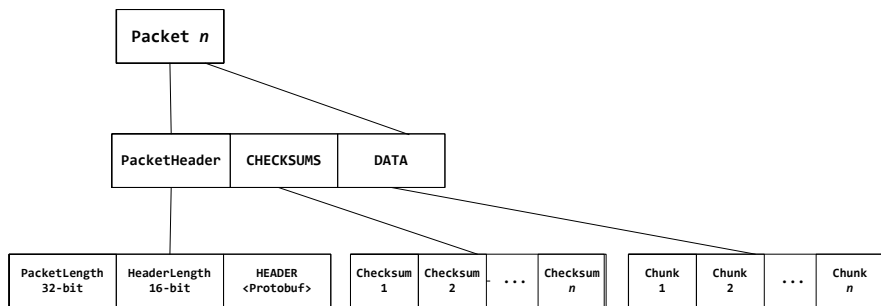


图 5-30 数据包结构

`DFSOutputStream.write()`方法可以将指定大小的数据写入数据流内部的一个缓冲区中，写入的数据会被切分成多个数据包，每个数据包又由一组校验块和这组校验块对应的校验和组成，默认数据包大小为 65536 字节，校验块大小为 512 字节，每个校验和都是校验块的 512 字节数据对应的校验值。这里的数据包大小、校验块大小是在 `computePacketChunkSize()`方法中定义的。

当 Client 写入的字节流数据达到一个数据包的长度时，`DFSOutputStream` 会构造一个 `Packet` 对象保存这个要发送的数据包。如果当前数据块中的所有数据包都发送完毕了，`DFSOutputStream` 会发送一个空的数据包标识数据块发送完毕。新构造的 `Packet` 对象会被放到 `DFSOutputStream.dataQueue` 队列中，由 `DFSOutputStream` 的内部线程类 `DataStreamer` 处理。

`DataStreamer` 线程会从 `dataQueue` 中取出 `Packet` 对象，然后通过底层 IO 流将这个 `Packet` 发送到数据流管道中的第一个 `Datanode` 上。发送完毕后，将 `Packet` 从 `dataQueue` 中移除，放入 `ackQueue` 中等待下游节点的确认消息。确认消息是由 `DataStreamer` 的内部线程类 `ResponseProcessor` 处理的。

`ResponseProcessor` 线程等待下游节点的响应 `ack`，判断 `ack` 状态码，如果是失败状态，则记录出错 `Datanode` 的索引 (`errorIndex & restartIndex`)，并设置错误状态位 (`hasError`)。如果 `ack` 状态是成功，则将数据包从 `ack` 队列中移除，整个数据包发送过程完成。数据包发送的完整流程如图 5-31 所示。

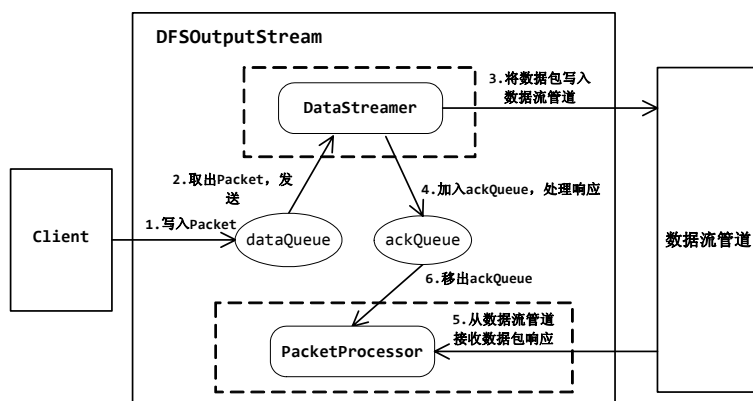


图 5-31 数据包发送流程图

如果在数据块发送过程中出现错误，那所有 `ackQueue` 队列中等待确认的 `Packet` 都会被重新放回 `dataQueue` 队列中重新发送。客户端会执行错误处理流程，将出现错误的 `Datanode` 从数据流管道中删除，然后向 `Namenode` 申请新的 `Datanode` 重建数据流管道。接着 `DataStreamer` 线程会从 `dataQueue` 队列中取出 `Packet` 重新发送。

1. `write()`方法

`DFSOutputStream.write()`方法继承自 `FSOutputSummer.write()`方法，用于向数据流管道中

写入指定大小的数据以及校验和,是客户端写数据操作的入口。`write()`方法会循环调用 `write1()`方法每次发送一个校验块数据,直到所有数据发送完毕。`write()`方法的代码如下:

```
public synchronized void write(byte b[], int off, int len)
    throws IOException {

    checkClosed();
    // 判断方法参数是否合法
    if (off < 0 || len < 0 || off > b.length - len) {
        throw new ArrayIndexOutOfBoundsException();
    }
    // 循环调用 write1(), 写数据, 每次写一个校验块
    for (int n=0;n<len;n+=write1(b, off+n, len-n)) {
    }
}
```

`write()`方法的调用流程如图 5-32 所示,我们依次介绍流程图中调用的各个方法。

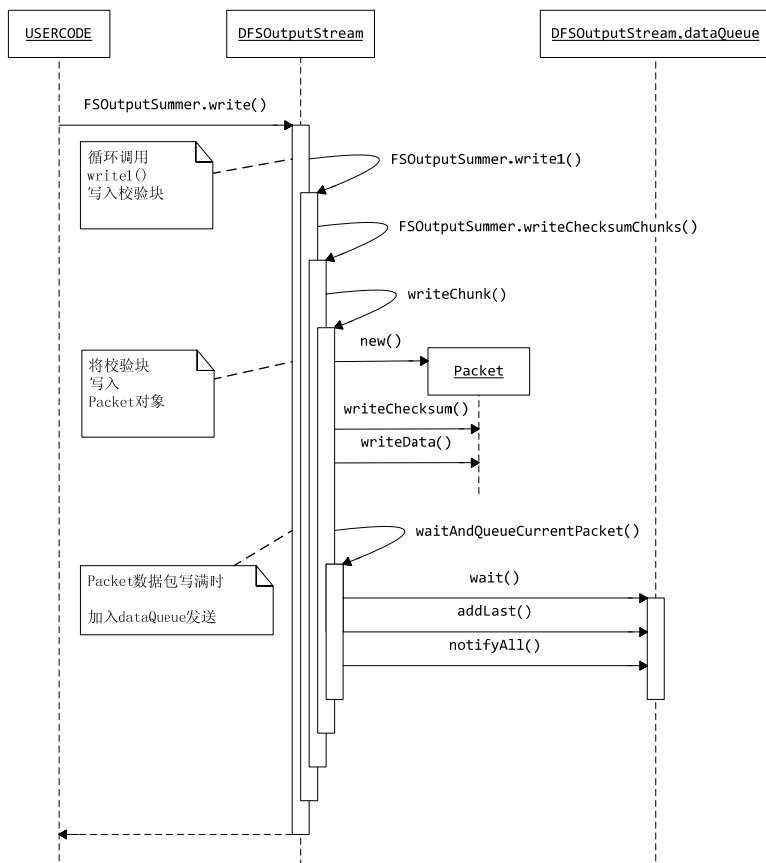


图 5-32 `write()`方法调用流程图

(1) write1()

`write1()`方法每次写入一个校验块数据，如果数据长度不足一个校验块，则写入 `buffer` 缓冲区。这个方法首先将数据写入 `buffer` 缓冲区，当 `buffer` 中的数据达到一个校验块（`chunk`）的大小时，则调用 `flushBuffer()`方法将缓存中的校验块写入底层 IO 流。如果 `buffer` 为空，并且写入的数据大于一个校验块的大小时，则调用 `writeChecksumChunk()`直接将校验块大小的数据写入 IO 流，不经过 `buffer` 缓存。`write()`和 `write1()`方法的代码如下：

```
private int write1(byte b[], int off, int len) throws IOException {
    // 如果 buffer 为空，并且当前写入数据大于一个 chunk 的大小，则调用 writeChecksumChunk() 直接将数
    据与校验和写入 IO 流
    if(count==0 && len>=buf.length) {
        final int length = buf.length;
        sum.update(b, off, length);
        writeChecksumChunk(b, off, length, false);
        return length;
    }

    // 否则先将数据写入 buffer 中
    int bytesToCopy = buf.length-count;
    bytesToCopy = (len<bytesToCopy) ? len : bytesToCopy;
    sum.update(b, off, bytesToCopy);
    System.arraycopy(b, off, buf, count, bytesToCopy);
    count += bytesToCopy;
    // 如果 buffer 满了，则调用 flushBuffer() 方法将缓存中的数据写入 IO 流
    if (count == buf.length) {
        flushBuffer();
    }
    return bytesToCopy;
}
```

这里无论是 `writeChecksumChunk()`还是 `flushBuffer()`方法，最终都调用了 `writeChunk()`方法将一个校验块以及校验块对应的校验和写入 IO 流。`writeChunk()`在 `FSOutputSummer` 中是一个抽象方法，最终由 `DFSOutputStream` 实现。

(2) writeChunk()

`writeChunk()`方法首先构造一个 `Packet` 对象保存数据包，然后将校验块数据以及校验和写入 `Packet` 对象中。当 `Packet` 对象写满时（每个数据包都可以写入 `maxChunks` 个校验块），则调用 `waitAndQueueCurrentPacket()`方法将当前 `Packet` 对象放入输出队列 `dataQueue` 中等待 `DataStream` 线程的处理。如果当前数据块中的所有数据都已经发送完毕，则发送一个空数据包标识所有数据已经发送完毕。

我们看一下 `writeChunk()`的代码实现。

```
protected synchronized void writeChunk(byte[] b, int offset, int len, byte[] checksum)
                                                                    throws IOException {
    dfsClient.checkOpen(); // 检查 DFSClient 状态
```

Hadoop 2.X HDFS 源码剖析

```
checkClosed();                // 检查 DFSOutputStream 状态

int cklen = checksum.length;   // 校验和数组长度
int bytesPerChecksum = this.checksum.getBytesPerChecksum(); // 每个校验和对应的数据长度
if (len > bytesPerChecksum) { // 输入的数据长度大于一个校验块 (chunk) 的大小时, 则抛出异常
    throw new IOException("writeChunk() buffer size is " + len +
        " is larger than supported bytesPerChecksum " +
        bytesPerChecksum);
}
if (checksum.length != this.checksum.getChecksumSize()) { // 校验数据长度错误, 抛出异常
    throw new IOException("writeChunk() checksum size is supposed to be " +
        this.checksum.getChecksumSize() +
        " but found to be " + checksum.length);
}

if (currentPacket == null) { // 当前数据包为空, 则构造一个新的数据包
    currentPacket = new Packet(packetSize, chunksPerPacket,
        bytesCurBlock);
}

// 将当前校验数据、校验块写入数据包中
currentPacket.writeChecksum(checksum, 0, cklen);
currentPacket.writeData(b, offset, len);
currentPacket.numChunks++;
bytesCurBlock += len;

// 如果当前数据包已经满了, 或者写满了一个数据块, 则将当前的数据包放入发送队列中
if (currentPacket.numChunks == currentPacket.maxChunks ||
    bytesCurBlock == blockSize) {
    waitAndQueueCurrentPacket();

    // 如果之前的 chunk 没有写满, 则当前 packet 只发送这个 append trunk。发送完后, 将 checksum 和
    appendChunk 重置
    if (appendChunk && bytesCurBlock%bytesPerChecksum == 0) {
        appendChunk = false;
        resetChecksumChunk(bytesPerChecksum);
    }

    // 恢复 packet 大小, 注意这里避免了越过数据块的边界
    if (!appendChunk) {
        int psize=Math.min((int) (blockSize-bytesCurBlock),dfsClient.getConf().writePacketSize);
        computePacketChunkSize(psize, bytesPerChecksum);
    }

    // 如果写满了一个数据块的长度, 则发送一个空 packet 作为标识, 表明发送了一个完整的数据块
    if (bytesCurBlock == blockSize) {
        currentPacket = new Packet(0, 0, bytesCurBlock);
        currentPacket.lastPacketInBlock = true;
    }
}
```

```

        currentPacket.syncBlock = shouldSyncBlock;
        waitAndQueueCurrentPacket();
        bytesCurBlock = 0;
        lastFlushOffset = 0;
    }
}
}

```

当写满一个数据包时，DFSOutputStream 会调用 waitAndQueueCurrentPacket()方法将数据包放入发送队列 dataQueue 中，等待 streamer 线程将数据包发送到数据流管道中。streamer 线程将 dataQueue 中的数据包发送出去之后，会将该数据包放入 ackQueue 队列中，只有当 ResponseProcessor 线程收到下游节点传来的 ack 消息之后，才会将数据包从 ackQueue 中移除。waitAndQueueCurrentPacket()方法的代码如下：

```

private void waitAndQueueCurrentPacket() throws IOException {
    synchronized (dataQueue) {
        try {
            while (!closed && dataQueue.size() + ackQueue.size() > MAX_PACKETS) {
                try {
                    // dataQueue 中数据包队列已经满了，这个时候在锁上等待，等待队列中有空余位置
                    dataQueue.wait();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
            checkClosed();
            queueCurrentPacket();
        } catch (ClosedChannelException e) {
        }
    }
}

private void queueCurrentPacket() {
    synchronized (dataQueue) {
        if (currentPacket == null) return;
        // 将当前 packet 加入 dataQueue 队列中
        dataQueue.addLast(currentPacket);
        lastQueuedSeqno = currentPacket.seqno;
        currentPacket = null;
        // 通知发送线程
        dataQueue.notifyAll();
    }
}
}

```

在 writeChunk()方法中有一点需要注意，当用户代码追加写（append）一个文件时，很有可能这个文件的最后一个数据块的最后一个校验块（chunk）并没有写满，这时需要先将文件的最后一个校验块写满，然后才能写入新的完整的校验块。在这种情况下，writeChunk()方法

会设置发送数据包的大小为填满文件最后一个校验块所需的字节数，将输出流的 `FSOutputSummer` 中 `buffer` 的大小也设置为填满文件最后一个校验块的字节数。这样第一次发送的数据包就可以将文件的最后一个校验块写满了。填满文件的最后一个校验块后，`writeChunk()` 方法重置发送数据包以及校验块的大小。这段代码在 `DataStreamer` 追加写操作的构造方法中可以看到。

```
private DataStreamer(LocatedBlock lastBlock, HdfsFileStatus stat,
    int bytesPerChecksum) throws IOException {
    // ....
    int usedInCksum = (int)(stat.getLen() % bytesPerChecksum); // 最后一个 chunk 中剩余的
数据
    int freeInCksum = bytesPerChecksum - usedInCksum; // 填满最后一个 chunk 所需的数据

    // 如果最后一个数据块是满的，则无法进行 append 操作，抛出异常，申请新的数据块
    if (freeInLastBlock == blockSize) {
        throw new IOException("The last block for file " +
            src + " is full.");
    }

    if (usedInCksum > 0 && freeInCksum > 0) {
        // 重置 packet 大小，将 packet 大小设置为 freeInCksum
        computePacketChunkSize(0, freeInCksum);
        // 重置 chunk 大小，将 chunk 大小设置为 freeInCksum，也就是 packet 中只有这一个补充的 chunk
        resetChecksumChunk(freeInCksum);
        appendChunk = true;
    } else {
        // ...
        computePacketChunkSize(Math.min(dfsClient.getConf().writePacketSize,
            freeInLastBlock), bytesPerChecksum);
    }
}
```

将数据包放入 `dataQueue` 队列中后，会由 `DataStreamer` 类负责发送与处理。`DataStreamer` 类的实现在下一小节中介绍。

2. DataStreamer 线程

我们知道 `DFSOutputStream` 的构造方法创建了一个 `DataStreamer` 对象，这个 `DataStreamer` 类是 `DFSOutputStream` 的内部线程类。它首先向 `Namenode` 申请一个新的数据块，然后建立写这个数据块的数据流管道（pipeline），最后 `DataStreamer` 从 `dataQueue` 队列中取出待发送数据包并通过数据流管道发送给 `Datanode`。每个数据包（packet）都有一个与之相关的序列号，当一个数据块中所有的数据包都发送完毕，并且获得了 `ACK` 消息后，`DataStreamer` 线程就会将当前数据块的数据流管道关闭。如果 `DFSOutputStream` 中还有数据需要发送，则 `DataStreamer` 会再次向 `Namenode` 申请分配新的数据块，并且提交上一个数据块。获取了新分配数据块的位置信息后，`DataStreamer` 会再次建立到新分配数据块的数据流管道，然后发送数据。

(1) 数据流管道的状态

`DataStream` 会将待发送数据包通过数据流管道发送给 `Datanode`，并在当前数据块写满时向 `Namenode` 申请分配新的数据块，然后更新数据流管道。所以 `DataStream` 类定义了 `nodes`、`storageTypes`、`storageIDs` 以及 `stage` 等字段，用于保存当前数据流管道的状态。同时，`DataStream` 还定义了 `setPipeline()` 方法用于更新上述字段。

```
// 数据块数据流管道中的 Datanode
private volatile DatanodeInfo[] nodes = null;
// 在 Datanode 上保存这个数据块的存储 (storage)
private volatile String[] storageIDs = null;
// 在 Datanode 上保存这个数据块存储的存储类型
private volatile StorageType[] storageTypes = null;
// 数据库对应的数据流管道的状态
private BlockConstructionStage stage;

private void setPipeline(DatanodeInfo[] nodes, StorageType[] storageTypes,
    String[] storageIDs) {
    this.nodes = nodes;
    this.storageTypes = storageTypes;
    this.storageIDs = storageIDs;
}
```

`DataStream` 使用 `stage` 字段记录了当前数据流管道的状态，数据流管道的状态定义在 `BlockConstructionStage` 类中，有如下几种。

- `PIPELINE_SETUP_CREATE`：写新文件时，数据流管道的初始状态。
- `PIPELINE_SETUP_APPEND`：追加写已有文件时，数据流管道的初始状态。
- `DATA_STREAMING`：数据流管道已经建立好，可以传输数据了。
- `PIPELINE_CLOSE`：数据块已经写满，数据流管道关闭。

数据流管道的状态迁移如图 5-33 所示。当 `DFSClient` 执行写新文件操作时，数据流管道的初始状态为 `PIPELINE_SETUP_CREATE`；当 `DFSClient` 执行追加写文件操作时，数据流管道的初始状态为 `PIPELINE_SETUP_APPEND`。

对于写新文件操作，`DataStream` 会调用 `nextBlockOutputStream()` 方法向 `Namenode` 申请分配新的数据块，然后构造这个新数据块的数据流管道。对于追加写文件操作，`DataStream` 会调用 `setupPipelineForAppendOrRecovery()` 方法打开已有的 HDFS 文件并返回这个文件最后一个数据块的位置信息，然后根据最后一个数据块的位置信息初始化数据流管道。

成功构造数据流管道后，`DataStream` 会调用 `initDataStreaming()` 方法将数据流管道状态改为 `DATA_STREAMING`，并调用 `setPipeline()` 记录数据流管道状态，然后就可以通过数据流管道发送数据包了。

当数据流管道将当前数据块写满后，会将数据流管道状态设置为 `PIPELINE_CLOSE`，然后向数据流管道发送一个空的数据包标识数据块已经写完。当 `DataStream` 确认收到这个空数据包的响应消息后，也就是数据流管道中的所有 `Datanode` 都成功地写入了数据块时，会调

用 `endBlock()` 方法关闭数据流管道，并将数据流管道状态设置为 `PIPELINE_SETUP_CREATE` 初始状态。然后 `DataStream` 会申请新的数据块，建立数据流管道，并写入数据，直到 `DataStream` 线程被关闭。

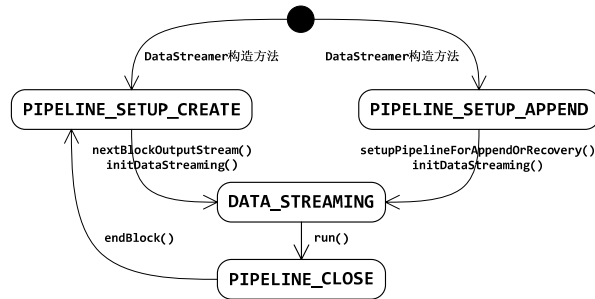


图 5-33 数据流管道状态迁移图

(2) 数据流管道的建立

`DataStream` 在将数据包发送到 `Datanode` 之前，首先要在 `Namenode` 的命名空间 (Namespace) 中分配数据块，建立写数据块的数据流管道。这些操作都是在 `DataStream.run()` 方法中触发的，首先调用 `nextBlockOutputStream()` 方法向 `Namenode` 申请分配新的数据块，然后建立到新分配数据块的输出流。接下来调用 `setPipeline()` 方法记录数据流管道信息 (包括存储数据的 `Datanode`，以及它们的 `storageIDs`)。最后调用 `initDataStreaming()` 启动 `Response Processor` 线程处理来自 `Datanode` 的响应信息，并将数据块构建状态 (`BlockConstructionStage`) 设置为 `DATA_STREAMING`。数据流管道的建立流程如图 5-34 所示。

我们看一下 `DataStream.run()` 方法，这里将与数据流管道建立无关的代码先省略，在后面的小节中介绍。

```

public void run() {
    while (!streamerClosed && dfsClient.clientRunning) {
        // responder 线程遇到故障，关闭 responder 线程
        // ... 暂时省略
        // 在 Namenode 上分配一个新的数据包
        if (stage == BlockConstructionStage.PIPELINE_SETUP_CREATE) {
            setPipeline(nextBlockOutputStream());
            initDataStreaming();
        } else if (stage == BlockConstructionStage.PIPELINE_SETUP_APPEND) {
            // ...
        }
        // .... 暂时省略
    }
}
  
```

下面我们分别介绍 `nextBlockOutputStream()`、`setPipeline()` 以及 `initDataStreaming()` 三个方法的实现。

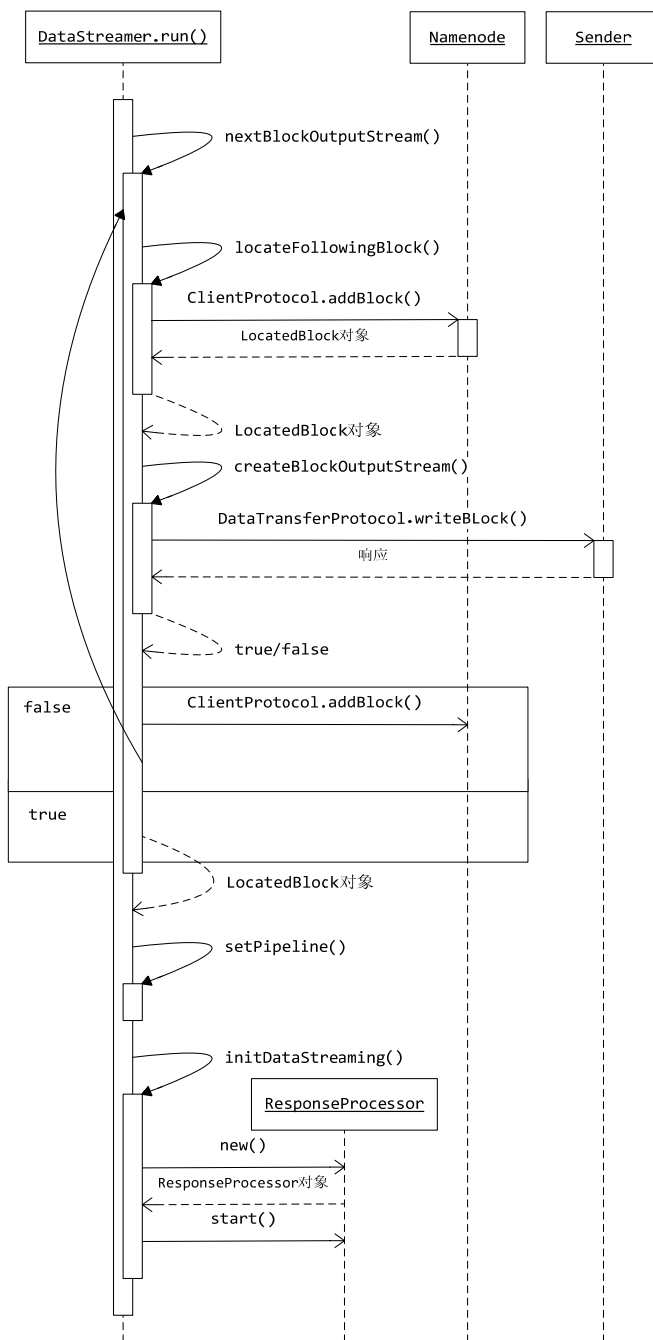


图 5-34 数据流管道建立流程图

nextBlockOutputStream()

`nextBlockOutputStream()`方法首先会调用 `locateFollowingBlock()`在 Namenode 上分配一个新的数据块，这个方法返回存储新数据块的 `Datanode` 位置信息。然后调用 `createBlockOutputStream()`方法创建到数据流管道中第一个 `Datanode` 的输出流，如果创建成功则返回 `true`，创建失败则返回 `false`。当创建输出流失败后，`nextBlockOutputStream()`方法会调用 `ClientProtocol.abandonBlock()`放弃这个数据块，并将这个 `Datanode` 加入故障节点队列中，以避免再次访问这个节点。然后重试申请操作，重试超过一定的次数后（由配置项 `dfs.client.block.write.retries` 配置，默认为 3 次），将抛出异常。`nextBlockOutputStream()`方法的代码如下：

```
private LocatedBlock nextBlockOutputStream() throws IOException {
    LocatedBlock lb = null; // 数据块存储的 Datanode 位置信息
    DatanodeInfo[] nodes = null; // 数据块存储在哪些 Datanode 上
    int count = dfsClient.getConf().nBlockWriteRetry; // 重试次数，默认为 3 次
    boolean success = false; // 数据流管道建立操作是否成功
    ExtendedBlock oldBlock = block;
    do {
        // ... 局部变量
        long startTime = Time.now();
        // 故障节点列表
        DatanodeInfo[] excluded =
            excludedNodes.getAllPresent(excludedNodes.asMap().keySet())
                .keySet()
                .toArray(new DatanodeInfo[0]);
        block = oldBlock;

        // 调用 locateFollowingBlock() 方法在 Namenode 上申请新的数据块
        lb = locateFollowingBlock(startTime,
            excluded.length > 0 ? excluded : null);

        block = lb.getBlock(); // 当前写的数据块
        block.setNumBytes(0); // 数据块大小，因为没有写，所以是 0
        bytesSent = 0; // 写入字节数，为 0
        accessToken = lb.getBlockToken();
        nodes = lb.getLocations(); // Datanode 信息

        // 调用 createBlockOutputStream() 方法建立到数据流管道中第一个 Datanode 的输出流
        success = createBlockOutputStream(nodes, 0L, false);

        if (!success) {
            // 创建输出流失败，则放弃数据块
            dfsClient.namenode.abandonBlock(block, fileId, src,
                dfsClient.clientName);
            block = null;
            // 将该 Datanode 放入故障节点列表中
            excludedNodes.put(nodes[errorIndex], nodes[errorIndex]);
        }
    }
```

```

    } while (!success && --count >= 0); // 进行重试操作

    // 如果重试不成功，则抛出异常
    if (!success) {
        throw new IOException("Unable to create new block.");
    }
    return lb;
}

```

`locateFollowingBlock()`方法会调用 `ClientProtocol.addBlock()`方法向 Namenode 申请分配一个新的数据块（`addBlock()`方法除了分配新的数据块之外，还会提交上一个数据块，然后返回存储新数据块的 Datanode 位置信息）。这里还要注意异常处理，对于 `NotReplicatedYetException`，也就是该文件的上一个提交数据块还没有达到 HDFS 最小副本数的情况，需要 Client 线程睡眠一段时间之后重试，等待上一个提交数据块有足够的副本。`locateFollowingBlock()`方法的代码如下：

```

private LocatedBlock locateFollowingBlock(long start,
    DatanodeInfo[] excludedNodes) throws IOException {
    int retries = dfsClient.getConf().nBlockWriteLocateFollowingRetry; // 重试次数，默认为 5 次
    long sleeptime = 400; // 等待时间为 400 毫秒
    while (true) {
        long localstart = Time.now();
        while (true) {
            try {
                // 调用 ClientProtocol.addBlock() 方法向 Namenode 申请分配一个新的数据块
                return dfsClient.namenode.addBlock(src, dfsClient.clientName,
                    block, excludedNodes, fileId, favoredNodes);
            } catch (RemoteException e) {
                IOException ue =
                    e.unwrapRemoteException(FileNotFoundException.class,
                                            AccessControlException.class,
                                            NSQuotaExceededException.class,
                                            DSQuotaExceededException.class,
                                            UnresolvedPathException.class);

                if (ue != e) {
                    throw ue; // no need to retry these exceptions
                }

                // 文件的上一个提交数据块没有达到 HDFS 系统指定的副本数，则等待一段时间，然后重试
                if (NotReplicatedYetException.class.getName().
                    equals(e.getClassName())) {
                    if (retries == 0) {
                        throw e;
                    } else {
                        --retries;
                        try {
                            Thread.sleep(sleeptime);
                        }

```

```
        sleeptime *= 2;
    } catch (InterruptedException ie) {
    }
}
} else {
    throw e;
}
}
}
}
```

`createBlockOutputStream()`方法会创建到数据流管道中第一个 `Datanode` 的输出流。这个实现比较简单，只需要通过调用 `TCP` 流式接口 `DataTransferProtocol.writeBlock()`方法向第一个 `Datanode` 发送写数据块请求即可。然后判断响应，如果响应状态为 `SUCCESS`，则返回成功；如果为失败，则抛出异常，由 `catch` 部分进行错误处理。`createBlockOutputStream()`方法的代码如下：

```
private boolean createBlockOutputStream(DatanodeInfo[] nodes, long newGS,
    boolean recoveryFlag) {
    while (true) {
        boolean result = false;
        DataOutputStream out = null;
        try {
            // ... 建立到数据流管道中第一个 Datanode 的 Socket 连接
            s = createSocketForPipeline(nodes[0], nodes.length, dfsClient);
            long writeTimeout = dfsClient.getDatanodeWriteTimeout(nodes.length);
            // ... 对 Socket 输出流进行包装
            out = new DataOutputStream(new BufferedOutputStream(unbufOut,
                HdfsConstants.SMALL_BUFFER_SIZE));
            blockReplyStream = new DataInputStream(unbufIn);

            // 构造 Sender 类，发送流式接口 DataTransferProtocol.writeBlock() 请求
            new Sender(out).writeBlock(block, accessToken, dfsClient.clientName,
                nodes, null, recoveryFlag? stage.getRecoveryStage() : stage,
                nodes.length, block.getNumBytes(), bytesSent, newGS, checksum,
                cachingStrategy.get());

            // 获取写数据块响应
            BlockOpResponseProto resp = BlockOpResponseProto.parseFrom(
                PBHelper.vintPrefixed(blockReplyStream));
            pipelineStatus = resp.getStatus();
            firstBadLink = resp.getFirstBadLink();

            // 如果数据流管道中有 Datanode 正在重启，则设置 checkRestart，并抛出异常
            if (PipelineAck.isRestartOOBStatus(pipelineStatus) &&
                restartingNodeIndex == -1) {
                checkRestart = true;
            }
        } catch (Exception e) {
            // ... 异常处理
        }
    }
}
```

```

        throw new IOException("A datanode is restarting.");
    }
    // 数据流管道建立不成功，则抛出异常
    if (pipelineStatus != SUCCESS) {
        if (pipelineStatus == Status.ERROR_ACCESS_TOKEN) {
            throw new InvalidBlockTokenException(
                "Got access token error for connect ack with firstBadLink as "
                + firstBadLink);
        } else {
            throw new IOException("Bad connect ack with firstBadLink as "
                + firstBadLink);
        }
    }
}

blockStream = out; // 将输出流赋值到 blockStream 类属性上
result = true; // 创建输出流成功
restartingNodeIndex = -1;
hasError = false;
}

// 异常响应部分...
return result;
}

```

下面我们看一下 `createBlockOutputStream()` 方法错误处理部分的代码。出现异常时，由于 `DataTransferProtocol.writeBlock()` 请求的响应会将异常信息携带回 `Client`，并记录出现异常的节点（通过 `firstBadLink` 变量），所以异常处理代码会找到异常节点在数据流管道中的位置，并记录在 `errorIndex` 变量中。在 `nextBlockOutputStream()` 方法中，会将这个错误的节点加入故障队列中，以后不再从故障队列中读取数据。

```

catch (IOException ie) {
    if (restartingNodeIndex == -1) {
        DFSClient.LOG.info("Exception in createBlockOutputStream", ie);
    }
    // 安全错误，重新获得 key
    if (ie instanceof InvalidEncryptionKeyException && refetchEncryptionKey>0){
        refetchEncryptionKey--;
        dfsClient.clearDataEncryptionKey();
        continue;
    }

    // 找到出现错误的节点，用 errorIndex 记录错误节点在 nodes[] 中的索引
    if (firstBadLink.length() != 0) {
        for (int i = 0; i < nodes.length; i++) {
            // NB: Unconditionally using the xfer addr w/o hostname
            if (firstBadLink.equals(nodes[i].getXferAddr())) {
                errorIndex = i;
                break;
            }
        }
    }
}

```

```
    }  
  }  
  } else {  
    assert checkRestart == false;  
    errorIndex = 0;  
  }  
  // 判断是否需要等待节点重启, 这里只会等待 local 节点重启  
  if (checkRestart && shouldWaitForRestart(errorIndex)) {  
    restartDeadline = dfsClient.getConf().datanodeRestartTimeout +  
      Time.now(); // 等待时长  
    restartingNodeIndex = errorIndex; // 等待重启的索引  
    errorIndex = -1;  
  }  
  hasError = true;  
  setLastException(ie); // 记录异常  
  result = false; // 失败  
} finally {  
  if (!result) {  
    // 失败时, 关闭输出流  
  }  
}  
return result;  
}
```

setPipeline()

这个方法非常简单, 就是保存了存储当前数据块的 **Datanode** 信息, 以及 **Datnaode** 上保存数据块的存储 (**storage**) 信息。

```
private void setPipeline(LocatedBlock lb) {  
  setPipeline(lb.getLocations(), lb.getStorageIDs());  
}  
private void setPipeline(DatanodeInfo[] nodes, String[] storageIDs) {  
  this.nodes = nodes;  
  this.storageIDs = storageIDs;  
}
```

initDataStreaming()

initDataStreaming() 直接启动 **ResponseProcessor** 线程, 并将数据块构建状态改为 **DATA_STREAMING**。至此, 数据流管道已经建立好, 可以进入传输数据状态了。

```
private void initDataStreaming() {  
  this.setName("DataStreamer for file " + src +  
    " block " + block);  
  response = new ResponseProcessor(nodes);  
  response.start();  
  stage = BlockConstructionStage.DATA_STREAMING;  
}
```


(3) DataStreamer.run()

`DataStream.run()`执行了 `DataStream` 的所有逻辑，是一个大的循环方法。`run()`方法首先判断 `DataStream` 线程是否关闭，以及 `DFSClient` 是否处于运行状态。这个方法比较特殊的是错误处理在循环的最前面，如果出现错误则关闭 `responder` 线程，并且调用 `processDatanodeError()`处理 `Datanode` 的错误。

处理完错误之后，`run()`方法会等待可以发送的数据包，这里的等待条件比较复杂。当 `DataStream` 和 `DFSClient` 都正常运行，当前没有错误发生，并且 `dataQueue` 队列中没有待发送的数据包时，线程会睡眠一段时间，等待其他线程发送数据包并将数据包放入 `dataQueue` 中。另一种情况是，`processDatanodeError()`需要睡眠一段时间之后再执行，例如等待重启的 `Datanode` 重启完毕。如果上述操作执行完之后还有错误，那么跳过循环剩余部分，继续处理错误；否则从 `dataQueue` 中取出一个待发送的数据包，将这个数据包发送出去。如果此时 `dataQueue` 队列仍然为空，则构造一个空的心跳数据包发送。

`run()`方法之后的操作就是通过调用 `nextBlockOutputStream()`方法建立数据流管道，获取到数据流管道中第一个 `Datanode` 的 IO 流，然后通过这个 IO 流发送数据包。发送数据包的流程包括如下几个步骤。

- 如果当前数据包是数据块中的最后一个（我们知道最后一个数据包是一个空包，用来标识数据块中的所有数据已经发送完毕），这个包只有在之前发送的所有数据包确认获得 `ack` 之后，才可以发送。
- 将数据包从 `dataQueue` 中移出，移入 `ackQueue` 队列，等待 `ack` 响应。
- 将数据包写入底层 IO 流中。
- 如果发送的数据包是最后一个空隔离包，则调用 `endBlock()`执行清理工作。

`run()`方法的代码如下所示，整个处理流程可以分为如下几步。

- 如果出现了错误，则关闭应答器线程（`response`）。
- 调用 `processDatanodeError()`方法处理数据节点错误（请参考下一小节错误处理的介绍）。
- 等待数据包。调用 `dataQueue.wait()`方法，等待其他线程将待发送数据包插入到 `dataQueue` 中。
- 向 `Namenode` 申请分配数据块，然后建立数据流管道。
- 发送数据包。

```
public void run() {
    // 循环处理 dataQueue
    while (!streamerClosed && dfsClient.clientRunning) {

        // 第 1 步：如果出现错误，则关闭 response 线程
        if (hasError && response != null) {
            try {
                response.close();
                response.join();
            } catch (Exception e) {
                // 这里可以添加对 response 关闭失败的处理
            }
        }

        // 等待数据包
        dataQueue.wait();

        // 取出数据包
        DataBlock b = dataQueue.poll();

        // 建立数据流管道
        DataStream ds = dfsClient.createStream(b);

        // 发送数据包
        ds.write(b.data);

        // 将数据包放入 ackQueue
        ds.ackQueue.add(b);

        // 等待 ack
        b.ackQueue.wait();

        // 将数据包放入 dataQueue
        dataQueue.add(b);

        // 清理工作
        ds.endBlock();
    }
}
```

Hadoop 2.X HDFS 源码剖析

```
        response = null;
    } catch (InterruptedException e) {
    }
}

Packet one;
try {
    // 第 2 步: 调用 processDatanodeError() 处理错误
    boolean doSleep = false;
    if (hasError && (errorIndex >= 0 || restartingNodeIndex >= 0)) {
        doSleep = processDatanodeError();
    }

    synchronized (dataQueue) {
        // 第 3 步: 在 dataQueue 上等待一个需要被发送的数据块
        long now = Time.now();
        while ((!streamerClosed && !hasError && dfsClient.clientRunning
            && dataQueue.size() == 0 &&
            (stage != BlockConstructionStage.DATA_STREAMING ||
            stage == BlockConstructionStage.DATA_STREAMING &&
            now - lastPacket < dfsClient.getConf().socketTimeout/2)) || doSleep ) {
            long timeout = dfsClient.getConf().socketTimeout/2 - (now-lastPacket);
            timeout = timeout <= 0 ? 1000 : timeout;
            timeout = (stage == BlockConstructionStage.DATA_STREAMING)?
                timeout : 1000;
            try {
                // 在 dataQueue 对象上等待
                dataQueue.wait(timeout);
            } catch (InterruptedException e) {
            }
            doSleep = false;
            now = Time.now();
        }
        // 出现错误, 则跳过后续内容
        if (streamerClosed || hasError || !dfsClient.clientRunning) {
            continue;
        }

        // 从 dataQueue 中取出要发送的 packet, 或者构造一个心跳包
        if (dataQueue.isEmpty()) {
            one = new Packet(); // heartbeat packet
        } else {
            one = dataQueue.getFirst(); // regular data packet
        }
    }
    assert one != null;

    // 第 4 步: 如果数据流管道处于初始状态, 则在 Namenode 上分配一个数据块, 并初始化数据流管道
```

```

if (stage == BlockConstructionStage.PIPELINE_SETUP_CREATE) {
    // 调用 nextBlockOutputStream() 方法初始化数据流管道
    setPipeline(nextBlockOutputStream());
    initDataStreaming();
} else if (stage == BlockConstructionStage.PIPELINE_SETUP_APPEND) {
    setupPipelineForAppendOrRecovery();
    initDataStreaming();
}

long lastByteOffsetInBlock = one.getLastByteOffsetBlock();
if (lastByteOffsetInBlock > blockSize) {
    throw new IOException("BlockSize " + blockSize +
        " is smaller than data size. " +
        " Offset of packet in block " +
        lastByteOffsetInBlock +
        " Aborting file " + src);
}

// 第5步：发送数据包
// 首先判断如果是数据块中的最后一个数据包，则等待之前所有数据包 ack 之后再发送这个数据包
if (one.lastPacketInBlock) {
    // wait for all data packets have been successfully acked
    synchronized (dataQueue) {
        while (!streamerClosed && !hasError &&
            ackQueue.size() != 0 && dfsClient.clientRunning) {
            try {
                // 等待 ackQueue 队列清空
                dataQueue.wait(1000);
            } catch (InterruptedException e) {
            }
        }
    }
    // 之前发送的数据包可能出现错误
    if (streamerClosed || hasError || !dfsClient.clientRunning) {
        continue;
    }
    // 在发送最后一个数据包之前将状态设置为 PIPELINE_CLOSE
    stage = BlockConstructionStage.PIPELINE_CLOSE;
}

// 将数据包从 dataQueue 队列移动到 ackQueue 队列中
synchronized (dataQueue) {
    // move packet from dataQueue to ackQueue
    if (!one.isHeartbeatPacket()) {
        dataQueue.removeFirst();
        ackQueue.addLast(one);
        dataQueue.notifyAll();
    }
}

```

```
}

// 将数据包写入底层 IO 流中
try {
    one.writeTo(blockStream);
    blockStream.flush();
} catch (IOException e) {
    // 如果写数据出现异常, 根据 HDFS-3398, 直接将数据流管道中的第一个数据节点标志为损坏节点
    tryMarkPrimaryDatanodeFailed();
    throw e;
}
lastPacket = Time.now();

// 更新已经发送的字节数
long tmpBytesSent = one.getLastByteOffsetBlock();
if (bytesSent < tmpBytesSent) {
    bytesSent = tmpBytesSent;
}

if (streamerClosed || hasError || !dfsClient.clientRunning) {
    continue;
}

// 如果是数据块中的最后一个数据包, 则等待这个数据包 ack, 然后调用 endBlock() 方法结束
if (one.lastPacketInBlock) {
    // 等待数据包 ack 消息
    synchronized (dataQueue) {
        while (!streamerClosed && !hasError &&
            ackQueue.size() != 0 && dfsClient.clientRunning) {
            dataQueue.wait(1000);
        }
    }
    if (streamerClosed || hasError || !dfsClient.clientRunning) {
        continue;
    }
    // 调用 endBlock() 方法结束这个数据块的传输
    endBlock();
}
if (progress != null) { progress.progress(); }

} catch (Throwable e) {
    if (restartingNodeIndex == -1) {
        DFSClient.LOG.warn("DataStreamer Exception", e);
    }
    if (e instanceof IOException) {
        setLastException((IOException)e);
    }
    hasError = true;
}
```

```

        if (errorIndex == -1 && restartingNodeIndex == -1) {
            streamerClosed = true;
        }
    }
}
closeInternal();
}

```

当 `DataStreamer` 线程完成一个数据块的写入操作后,会调用 `endBlock()`方法关闭到这个数据块的数据流管道,包括关闭 `responder` 线程,关闭数据流管道的输出流 `blockStream` 以及输入流 `blockReplyStream`,然后更新数据流管道状态为 `PIPELINE_SETUP_CREATE`。

```

private void endBlock() {
    closeResponder();
    closeStream();
    setPipeline(null, null, null);
    stage = BlockConstructionStage.PIPELINE_SETUP_CREATE;
}

```

(4) 错误处理

本节介绍 `DataStreamer` 错误处理部分,主要是由 `processDatanodeError()`方法实现的。在 `DataStreamer` 发送数据包的过程中,由 `hasError`、`errorIndex` 和 `restartingNodeIndex` 这三个变量记录错误信息,它们分别表明数据流管道是否出现错误、数据流管道中错误的 `Datanode` 索引、数据流管道中需要重启的 `Datanode` 索引。

在发送数据包的过程中,可能出现如下错误。

- 在建立数据流管道的过程中:在 `createBlockOutputStream()`方法中,调用 `DataTransferProtocol.writeBlock()`请求时,下游 `Datanode` 可能出现异常,并随着 `writeBlock()`的响应带回。在异常处理代码中,对 `hasError`、`errorIndex` 和 `restartingNodeIndex` 这三个变量赋值。
- 数据包发送完成后:下游节点会对每个数据包进行 `ack` 确认。`ack` 确认消息中就会携带出现故障的 `Datanode` 信息,也就是在 `ResponseProcessor.run()`方法处理 `ack` 消息时,异常处理代码会对 `hasError`、`errorIndex` 和 `restartingNodeIndex` 这三个变量赋值。
- 在 `DataStreamer.run()`中,通过底层 IO 流发送数据包时会出现异常(这个时候,由于没有下游节点返回的消息,所以直接将数据管道流中的第一个节点标识为错误节点)。

出现错误之后的错误处理是由 `processDatanodeError()`方法实现的,它的流程可以分为以下三个部分。

- 关闭当前 IO 流。
- 将 `ackQueue` 队列中的元素移动到 `dataQueue` 中重新发送。
- 重新初始化数据流管道。

调用 `processDatanodeError()`方法时,数据流管道已经出现了错误,所以错误处理的第 1 步就是关闭数据流管道,包括下行到 `Datanode` 的输出流 `blockStream`,以及 `Datanode` 返回 `ack`

的输入流 `blockReplyStream`。第 2 步是 `processDatanodeError()` 方法将 `ackQueue` 中等待确认的数据包全部加回 `dataQueue`，因为这些包可能在发送过程中就出现了异常，所以需要重新发送。第 3 步是 `processDatanodeError()` 方法最后会调用 `setupPipelineForAppendOrRecovery()` 方法重置数据流管道。需要注意的是，如果数据流管道的状态是 `PIPELINE_CLOSE`，也就是数据块中除了最后一个空的隔离数据包外，所有数据包都已经发送完毕，并且成功接收了 `ack`。这个时候就不需要重置数据流管道了，由于底层 IO 流已经不存在，这时直接调用 `endBlock()` 进行清理即可。否则，调用 `initDataStreaming()` 重置数据流管道，启动 `ResponseProcessor` 线程，并将数据流管道状态设置为 `DATA_STREAMING`。`processDatanodeError()` 方法的代码如下：

```
private boolean processDatanodeError() throws IOException {
    // 第 1 步：关闭底层上行、下行 IO 流
    closeStream();

    // 第 2 步：将 ackQueue 中的数据包移入 dataQueue 中
    synchronized (dataQueue) {
        dataQueue.addAll(0, ackQueue);
        ackQueue.clear();
    }

    // 如果在同一个数据包上错误恢复的次数超过 5 次，则直接关闭 streamer，停止错误恢复流程
    if (lastAackedSeqnoBeforeFailure != lastAackedSeqno) {
        lastAackedSeqnoBeforeFailure = lastAackedSeqno;
        pipelineRecoveryCount = 1;
    } else {
        if (++pipelineRecoveryCount > 5) {
            lastException.set(new IOException("Failing write. Tried pipeline " +
                "recovery 5 times without success."));
            streamerClosed = true;
            return false;
        }
    }

    // 第 3 步：调用 setupPipelineForAppendOrRecovery() 方法重置数据流管道
    boolean doSleep = setupPipelineForAppendOrRecovery();

    if (!streamerClosed && dfsClient.clientRunning) {
        // 如果当前数据块的所有数据包都已经发送完毕，并且收到 ack，则直接调用 endBlock() 结束当前包的传输
        if (stage == BlockConstructionStage.PIPELINE_CLOSE) {
            synchronized (dataQueue) {
                Packet endOfBlockPacket = dataQueue.remove(); // remove the end of block packet
                assert endOfBlockPacket.lastPacketInBlock;
                assert lastAackedSeqno == endOfBlockPacket.seqno - 1;
                lastAackedSeqno = endOfBlockPacket.seqno;
                dataQueue.notifyAll();
            }
            endBlock();
        } else {
```

```

        // 重置数据流管道, 启动新的 responder 线程, 并将状态设置为 DATA_STREAMING
        initDataStreaming();
    }
}
return doSleep;
}

```

initDataStreaming()&endBlock()

这里我们看一下 `initDataStreaming()` 方法和 `endBlock()` 方法的实现。`initDataStreaming()` 方法直接启动新的 `ResponseProcessor` 线程, 并将状态设置为 `DATA_STREAMING`。`endBlock()` 则关闭 `responder` 线程, 关闭底层 IO 流, 将状态设置为 `PIPELINE_SETUP_CREATE`。

```

private void initDataStreaming() {
    this.setName("DataStreamer for file " + src +
        " block " + block);
    response = new ResponseProcessor(nodes);
    response.start();
    stage = BlockConstructionStage.DATA_STREAMING;
}

private void endBlock() {
    this.setName("DataStreamer for file " + src);
    closeResponder();
    closeStream();
    setPipeline(null, null);
    stage = BlockConstructionStage.PIPELINE_SETUP_CREATE;
}

```

setupPipelineForAppendOrRecovery()

`setupPipelineForAppendOrRecovery()` 方法主要应用在两种情况下: ① `append` 操作时用于创建数据流管道; ② 数据流管道写数据出现异常时, 进行恢复操作。

恢复操作有三个目的: ① 等待重启的 `Datanode` 启动; ② 将错误的 `Datanode` 从数据流管道中删除, 将新的 `Datanode` 添加到数据流管道中; ③ 最后更新 `Namenode` 命名空间中数据块的时间戳, 这样异常 `Datanode` 上的过期数据块就可以被删除了。

`setupPipelineForAppendOrRecovery()` 方法的执行流程分为如下几步。

- 处理重启的 `Datanode`, 这里的处理方式是线程睡眠一段时间 (默认为 4 秒), 等待 `Datanode` 重启。如果睡眠时间过长, 超过 `restartDeadline`, 那么将这个重启的 `Datanode` 标志为错误节点。
- 处理异常的 `Datanode`, 处理方式是从数据流管道中移除异常的 `Datanode`。
- 如果满足了替换异常 `Datanode` 的条件, 则调用 `addDatanode2ExistingPipeline()` 方法在数据流管道中添加一个新的 `Datanode`。
- 调用 `ClientProtocol.updateBlockForPipeline()` 更新数据块的时间戳, 这样异常 `Datanode` 上的时间戳错误的过期数据块就可以被删除了。

- 如果数据流管道恢复成功，则更新 Namenode 命名空间中数据块的时间戳，同时更新当前 Client 侧缓存的数据块信息的时间戳。

setupPipelineForAppendOrRecovery()方法的代码如下：

```
private boolean setupPipelineForAppendOrRecovery() throws IOException {
    // 如果数据流管道中没有 Datanode，则直接停止 DataStreamer
    if (nodes == null || nodes.length == 0) {
        setLastException(new IOException(msg));
        streamerClosed = true;
        return false;
    }

    boolean success = false;
    long newGS = 0L;
    while (!success && !streamerClosed && dfsClient.clientRunning) {
        // 循环等待需要重启的 Datanode 重启，直到超时
        if (restartingNodeIndex >= 0) {
            // 默认等待 4 秒
            long delay = Math.min(dfsClient.getConf().datanodeRestartTimeout,
                                   4000L);
            try {
                Thread.sleep(delay);
            } catch (InterruptedException ie) {
                lastException.set(new IOException("Interrupted while waiting for " +
                                                    "datanode to restart. " + nodes[restartingNodeIndex]));
                streamerClosed = true;
                return false;
            }
        }
        boolean isRecovery = hasError;

        // 处理错误节点
        if (errorIndex >= 0) {
            StringBuilder pipelineMsg = new StringBuilder();
            // 如果数据流管道中只有一个节点，则关闭 DataStreamer
            if (nodes.length <= 1) {
                lastException.set(new IOException("All datanodes " + pipelineMsg
                                                    + " are bad. Aborting..."));
                streamerClosed = true;
                return false;
            }
            // 将失败节点加入失败队列中
            failed.add(nodes[errorIndex]);

            // 从数据流管道数据节点中移除失败节点
            DatanodeInfo[] newnodes = new DatanodeInfo[nodes.length-1];
            System.arraycopy(nodes, 0, newnodes, 0, errorIndex);
            System.arraycopy(nodes, errorIndex+1, newnodes, errorIndex,
```



```

        newnodes.length-errorIndex);

    final String[] newStorageIDs = new String[newnodes.length];
    System.arraycopy(storageIDs, 0, newStorageIDs, 0, errorIndex);
    System.arraycopy(storageIDs, errorIndex+1, newStorageIDs, errorIndex,
        newStorageIDs.length-errorIndex);
    // 重置数据流管道
    setPipeline(newnodes, newStorageIDs);

    // 由于删除了节点，所以需要更新 restartingNodeIndex
    if (restartingNodeIndex >= 0) {
        if (errorIndex > restartingNodeIndex) {
            restartingNodeIndex = -1;
        } else if (errorIndex < restartingNodeIndex) {
            // restartingNodeIndex 需要减1，因为错误节点已经删除了
            restartingNodeIndex--;
        } else {
            // this shouldn't happen...
            assert false;
        }
    }

    if (restartingNodeIndex == -1) {
        hasError = false;
    }
    lastException.set(null);
    errorIndex = -1;
}

// 如果满足了替换异常 Datanode 的条件，则在数据流管道中添加一个新的 Datanode
if (dfsClient.dtpReplaceDatanodeOnFailure.satisfy(blockReplication,
    nodes, isAppend, isHflushed)) {
    addDatanode2ExistingPipeline();
}

// 获得一个新的时间戳
LocatedBlock lb = dfsClient.namenode.updateBlockForPipeline(block,
dfsClient.clientName);
newGS = lb.getBlock().getGenerationStamp();
accessToken = lb.getBlockToken();

// set up the pipeline again with the remaining nodes
if (failPacket) {
    // 测试代码
} else {
    // 更新数据流管道之后，重建数据流管道的 IO 流
    success = createBlockOutputStream(nodes, newGS, isRecovery);
}

```

```
if (restartingNodeIndex >= 0) {
    assert hasError == true;
    if (errorIndex == restartingNodeIndex) {
        errorIndex = -1;
    }
    // 超过 deadLine 之后, 就将重启节点设置为错误节点
    if (Time.now() < restartDeadline) {
        continue; // with in the deadline
    }
    restartDeadline = 0;
    int expiredNodeIndex = restartingNodeIndex;
    restartingNodeIndex = -1;
    if (errorIndex == -1) {
        errorIndex = expiredNodeIndex;
    }
}
} // while

if (success) {
    // 更新 Namenode 命名空间中数据块的时间戳
    ExtendedBlock newBlock = new ExtendedBlock(
        block.getBlockPoolId(), block.getBlockId(), block.getNumBytes(), newGS);
    dfsClient.namenode.updatePipeline(dfsClient.clientName, block, newBlock,
        nodes, storageIDs);
    // 更新客户端侧数据块的时间戳
    block = newBlock;
}
return false; // do not sleep, continue processing
}
```

在满足什么样的条件下, 会用新的 **Datanode** 替换数据流管道中异常的 **Datanode** 呢? 根据配置, 当当前数据节点的数目小于所需要的副本数目除 2, 以及在 **append/hflushed** 操作下, 当前数据节点的数目小于所需要的副本数目时, 调用 **addDatanode2ExistingPipeline()** 方法将新的数据节点添加到当前的数据流管道中。

下面我们来看一下 **addDatanode2ExistingPipeline()** 方法的实现。这个方法首先调用 **ClientProtocol.getAdditionalDatanode()** 获取新的 **Datanode**, 并将新获取的 **Datanode** 添加到数据流管道中。然后判断是否需要将正确节点上已经写入的部分数据块复制 (**transfer**) 到新添加的数据节点上。复制操作是由 **transfer()** 方法实现的, 底层调用了 **TCP** 流式接口 **DataTransferProtocol.transfer()** 方法, 将源数据节点上指定数据块的内容传输到目标节点上。

这里是否需要进行数据块复制操作的条件如下。

- 数据流管道处于初始状态 (**set up**)。
 - **append** 操作: 复制保存的副本, 这个副本的状态可能是 **RBW** 或者 **FINALIZED**。
 - **create** 操作: 如果没有数据写入, 则不复制; 如果有数据写入, 则复制。

- 在数据流管道写入（streaming）过程中出现错误。

无论是 append 还是 create 操作，都需要复制 RBW 状态的数据块。

- 关闭状态时错误（close）

无论是 append 还是 create，都不做处理，因为数据都已经成功传输了，这时交给 Namenode 来进行完整数据块的冗余备份操作即可。

下面我们看一下 addDatanode2ExistingPipeline()方法的代码。

```
private void addDatanode2ExistingPipeline() throws IOException {
    if (DataTransferProtocol.LOG.isDebugEnabled()) {
        DataTransferProtocol.LOG.debug("lastAckedSeqno = " + lastAckedSeqno);
    }
    // 在没有数据写入的情况下，什么也不用做
    if (!isAppend && lastAckedSeqno < 0
        && stage == BlockConstructionStage.PIPELINE_SETUP_CREATE) {
        return;
    } else if (stage == BlockConstructionStage.PIPELINE_CLOSE
        || stage == BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {
        // 数据流管道关闭，也不用进行操作
        return;
    }

    // 通过调用 ClientProtocol.getAdditionDatanode() 获取新的 Datanode，添加到数据流管道中
    final DatanodeInfo[] original = nodes;
    final LocatedBlock lb = dfsClient.namenode.getAdditionalDatanode(
        src, fileId, block, nodes, storageIDs,
        failed.toArray(new DatanodeInfo[failed.size()]));
    setPipeline(lb);
    // 找到这个新的 Datanode 在 nodes[] 中的索引
    final int d = findNewDatanode(original);

    // 调用 DataTransferProtocol 将数据块复制到另一个数据节点上
    final DatanodeInfo src = d == 0? nodes[1]: nodes[d - 1];
    final DatanodeInfo[] targets = {nodes[d]};
    transfer(src, targets, lb.getBlockToken());
}
```

3. ResponseProcessor 线程

对于 DataStreamer 发出的每一个数据包，数据流管道中的 Datanode 都会发送 ack 响应给客户端。ResponseProcessor 线程就是处理 ack 响应的线程类，类似于 Datanode 上的 PacketResponder 线程。

ResponseProcessor 线程的处理逻辑比较简单，它从数据流管道下游节点的输入流中读入响应消息。然后判断响应状态，如果下游数据节点执行写入数据包失败，则通过 ack 消息中

的应答码记录错误节点（`errorIndex`），并设置错误标志位（`hashError`）。最后会在 `DataStream.run()` 方法中调用 `processDatanodeError()` 处理这个错误信息。如果下游节点写入数据包成功，则把当前数据包信息从 `ackQueue` 中移除。至此，一个数据包就成功地写入了。

这里需要注意的是，在 `DataStream` 的出现错误部分都是直接抛出异常，然后在异常捕获代码部分设置错误信息。这种设计方式值得积累。

`ResponseProcessor.run()` 方法的代码如下：

```
public void run() {
    PipelineAck ack = new PipelineAck();

    while (!responderClosed && dfsClient.clientRunning && !isLastPacketInBlock) {
        try {

            // 从输入流中读取响应 ack
            ack.readFields(blockReplyStream);
            long seqno = ack.getSeqno();
            for (int i = ack.getNumOfReplies()-1; i >=0 && dfsClient.clientRunning; i--) {
                final Status reply = ack.getReply(i);
                // 处理重启的 Datanode
                if (PipelineAck.isRestartOOBStatus(reply) &&
                    shouldWaitForRestart(i)) {
                    restartDeadline = dfsClient.getConf().datanodeRestartTimeout +
                        Time.now();
                    setRestartingNodeIndex(i);
                    String message = "A datanode is restarting: " + targets[i];
                    DFSClient.LOG.info(message);
                    throw new IOException(message);
                }
                // 处理错误的数据节点
                if (reply != SUCCESS) {
                    setErrorIndex(i); // first bad datanode
                    throw new IOException("Bad response " + reply +
                        " for block " + block +
                        " from datanode " +
                        targets[i]);
                }
            }
            // 心跳消息的响应，则不用特别处理
            if (seqno == Packet.HEART_BEAT_SEQNO) { // a heartbeat ack
                continue;
            }

            // 数据流管道成功写入数据包的消息
            Packet one;
            synchronized (dataQueue) {
                one = ackQueue.getFirst();
```

```

    }
    if (one.seqno != seqno) {
        throw new IOException("ResponseProcessor: Expecting seqno " +
                                " for block " + block +
                                one.seqno + " but received " + seqno);
    }
    isLastPacketInBlock = one.lastPacketInBlock;

    // 以下都是成功处理的代码
    // 设置接收的字节数
    block.setNumBytes(one.getLastByteOffsetBlock());
    // 从 ackQueue 中取出对应的数据包
    synchronized (dataQueue) {
        lastAckedSeqno = seqno;
        ackQueue.removeFirst();
        dataQueue.notifyAll();
    }
} catch (Exception e) {
    // 出现异常的部分都是直接将异常抛出，然后在 catch 语句中设置异常状态位
    if (!responderClosed) {
        if (e instanceof IOException) {
            setLastException((IOException)e);
        }
        hasError = true;
        tryMarkPrimaryDatanodeFailed();
        synchronized (dataQueue) {
            dataQueue.notifyAll();
        }
        if (restartingNodeIndex == -1) {
            DFSClient.LOG.warn("DFSOutputStream ResponseProcessor exception "
                                + " for block " + block, e);
        }
        responderClosed = true;
    }
}
}
}

```

5.4.3 追加写操作

客户端除了可以执行写新文件的操作外，还可以打开一个已有的文件并执行追加写操作。`DistributedFileSystem.append()`方法就是用于打开一个已有的 HDFS 文件，并获取追加写操作的 `HdfsDataOutputStream` 对象。

如图 5-35 所示，`DistributedFileSystem.append()`方法调用了 `DFSClient.callAppend()`方法获取输出流对象。`callAppend()`方法首先通过 `ClientProtocol.append()`方法获取文件最后一个数据块的位置信息，如果文件的最后一个数据块已经写满则返回 `null`。然后 `callAppend()`方法会调

用 `DFSOutputStream.newStreamForAppend()` 方法创建到文件最后一个数据块的输出流对象。获取文件租约，并将新构建的 `DFSOutputStream` 包装为 `HdfsDataOutputStream` 对象，然后返回。

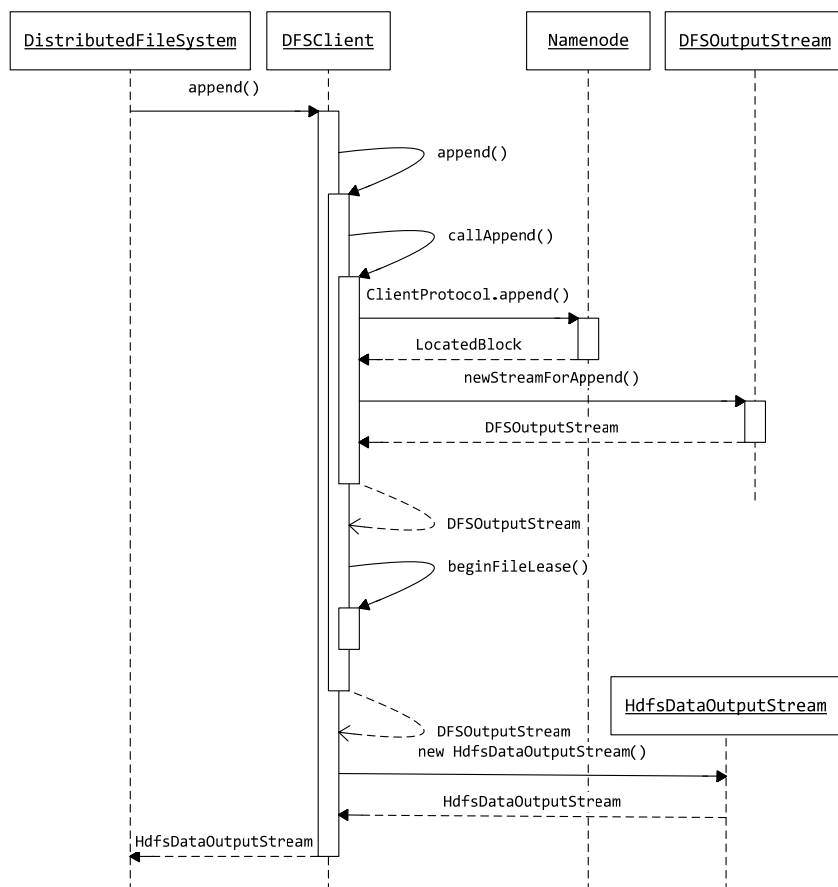


图 5-35 DistributedFileSystem.append()方法流程图

下面我们看一下 `DFSOutputStream.newStreamForAppend()` 方法的实现。这个方法调用了 `DFSOutputStream` 中构造 `append` 输出流的私有构造方法。这个构造方法会判断文件最后一个数据块是否写满，如果没有写满则构造用于追加写数据块的 `DataStreamer` 对象，然后写满这个数据块；如果文件最后一个数据块已经写满，则构造普通的 `DataStreamer` 对象向 `Namenode` 申请新的数据块并写入数据。`DFSOutputStream` 用于 `append` 操作的构造方法代码如下：

```

private DFSOutputStream(DFSClient dfsClient, String src,
    Progressable progress, LocatedBlock lastBlock, HdfsFileStatus stat,
    DataChecksum checksum) throws IOException {
    this(dfsClient, src, progress, stat, checksum); // 初始化一些字段
    // ...
    // 最后一个数据块没有写满
  
```

```

if (lastBlock != null) {
    // 构造执行 append 操作的 DataStreamer 线程
    bytesCurBlock = lastBlock.getBlockSize();
    streamer = new DataStreamer(lastBlock, stat, bytesPerChecksum, traceSpan);
} else {
    // 否则构造正常的 DataStreamer 线程
    computePacketChunkSize(dfsClient.getConf().writePacketSize, bytesPerChecksum);
    streamer = new DataStreamer(stat, traceSpan);
}
this.fileEncryptionInfo = stat.getFileEncryptionInfo();
}

```

如果文件的最后一个数据块写满了，则构造正常的 `DataStreamer` 对象（数据流管道状态为 `PIPELINE_SETUP_CREATE`）。向文件追加写数据时，这个 `DataStreamer` 对象会调用 `nextBlockOutputStream()` 方法向 `Namenode` 申请新的数据块并构造数据流管道。这部分内容我们在写操作小节中已经介绍了。而执行 `append` 操作的 `DataStreamer` 对象（数据流管道状态为 `PIPELINE_SETUP_APPEND`）会调用 `setupPipelineForAppendOrRecovery()` 方法建立到文件最后一个数据块的数据流管道（请参考 `DataStreamer` 线程的错误处理小节），然后执行追加写操作。写满这个数据块后，`DataStreamer` 对象会恢复为正常（数据流管道状态为 `PIPELINE_SETUP_CREATE`）。`DataStreamer` 的构造方法如下：

```

private DataStreamer(LocatedBlock lastBlock, HdfsFileStatus stat,
    int bytesPerChecksum, Span span) throws IOException {
    // 将 append 标识符设置为 true
    isAppend = true;
    // 将数据流管道状态设置为 PIPELINE_SETUP_APPEND
    stage = BlockConstructionStage.PIPELINE_SETUP_APPEND;
    // ...
    if (usedInCksum > 0 && freeInCksum > 0) {
        // 如果最后一个数据块的最后一个校验块中还有空间，则先填满这个校验块
        computePacketChunkSize(0, freeInCksum);
        setChecksumBufSize(freeInCksum);
        appendChunk = true;
    } else {
        // 如果最后一个数据块的大小小于发送数据包大小，则更改发送数据包大小
        computePacketChunkSize(Math.min(dfsClient.getConf().writePacketSize,
            freeInLastBlock), bytesPerChecksum);
    }

    // 保存最后一个数据块的数据流管道状态
    setPipeline(lastBlock);
    errorIndex = -1;
    if (nodes.length < 1) {
        throw new IOException("Unable to retrieve blocks locations " +
            " for last block " + block +
            " of file " + src);
    }
}

```

需要注意的是，如果文件最后一个数据块的最后一个校验块还没有写满，则在下一次写入操作时，先将这个校验块写满。`DataStreamer` 采用的方法是更改下一次发送的数据包大小，让这个数据包只包含一个校验块，并且将这个校验块的大小设置为填满文件最后一个数据块的最后一个校验块所需数据的大小。另一种情况是，文件最后一个数据块在校验块的边界上，但是剩余的空间小于 `DataStreamer` 发送数据包的大小，那么 `DataStreamer` 需要更改下一次发送数据包的大小为填满这个数据块所需数据的大小，这样才能在下一次写数据时将最后一个数据块填满。`DataStreamer.run()`在发送后续数据包时，会先调用 `computePacketChunkSize()`方法将数据包大小以及校验块大小恢复为默认的。

5.4.4 租约相关

租约是 HDFS 中一个很重要的概念，是 Namenode 给予租约持有者（`LeaseHolder`）（一般是客户端）在规定时间内拥有文件权限（写文件）的合同。

客户端写文件时需要先从租约管理器（`LeaseManager`）中申请一个租约，成功申请之后客户端就成为了租约持有者，也就拥有了对该 HDFS 文件的独占权限，其他客户端在该租约有效时无法打开这个 HDFS 文件进行操作。`Namenode` 的租约管理器会定期检查它维护的租约是否过期，如果有过期的租约，租约管理器会执行租约恢复机制关闭 HDFS 文件，所以持有租约的客户端需要定期更新租约（`renew`）。当客户端完成了对文件的写操作，关闭文件时，必须在租约管理器中释放租约。

无论是 `DFSClient.create()`还是 `DFSClient.append()`方法创建 `DFSOutputStream` 对象时，都会调用 `beginFileLease()`方法获取 HDFS 文件租约，并开始执行租约更新操作。`beginFileLease()`方法调用了 `LeaseRenewer.put()`方法将打开的 HDFS 文件以及输出流作为一个记录，放入 `DFSClient` 的租约管理器 `LeaseRenewer` 对象中。`beginFileLease()`方法的代码如下：

```
/** 获取 HDFS 文件租约，并开始执行租约更新操作*/
private void beginFileLease(final long inodeId, final DFSOutputStream out)
    throws IOException {
    getLeaseRenewer().put(inodeId, out, this);
}

public LeaseRenewer getLeaseRenewer() throws IOException {
    return LeaseRenewer.getInstance(authority, ugi, this);
}
```

`LeaseRenewer.put()`方法会构造一个租约检查线程，然后启动这个线程，这个线程会调用 `LeaseRenewer.run()`方法执行租约检查操作。`LeaseRenewer.put()`方法的代码如下：

```
synchronized void put(final long inodeId, final DFSOutputStream out,
    final DFSClient dfsc) {
    if (dfsc.isClientRunning()) {
        if (!isRunning() || isRenewerExpired()) {
```



```

final int id = ++currentId;
// 构造并启动租约检查线程
daemon = new Daemon(new Runnable() {
    public void run() {
        try {
            LeaseRenewer.this.run(id);
        } catch (InterruptedException e) {
        } finally {
            // ...
        }
    }
});
daemon.start();
}
dfsc.putFileBeingWritten(inodeId, out);
emptyTime = Long.MAX_VALUE;
}
}

```

LeaseRenewer.run()方法的实现很简单，如果当前时间和上一次租约更新时间的间隔大于租约软超时时间（1 分钟，不可配置）的一半，则调用 **renew()**方法更新租约，**renew()**方法底层调用了 **ClientProtocol.renewLease()**方法向 Namenode 发起更新租约请求。**LeaseRenewer.run()**方法的代码如下：

```

private void run(final int id) throws InterruptedException {
    for(long lastRenewed = Time.now(); !Thread.interrupted();
        Thread.sleep(getSleepPeriod())) {
        final long elapsed = Time.now() - lastRenewed;
        if (elapsed >= getRenewalTime()) {
            try {
                renew();
                lastRenewed = Time.now();
            } catch (SocketTimeoutException ie) {
                synchronized (this) {
                    while (!dfsclients.isEmpty()) {
                        dfsclients.get(0).abort();
                    }
                }
                break;
            } catch (IOException ie) {
                // ...
            }
        }
    }

    synchronized(this) {
        if (id != currentId || isRenewerExpired()) {
            return;
        }
    }
}

```

```

    if (!clientsRunning() && emptyTime == Long.MAX_VALUE) {
        emptyTime = Time.now();
    }
}
}
}
}

```

5.4.5 关闭输出流

介绍完了 DFSOutputStream 写入数据的操作后，我们就来学习 DFSOutputStream 的关闭操作。DFSOutputStream.close()方法实现了 DFSOutputStream 的关闭操作，图 5-36 给出了 close() 方法的调用流程。

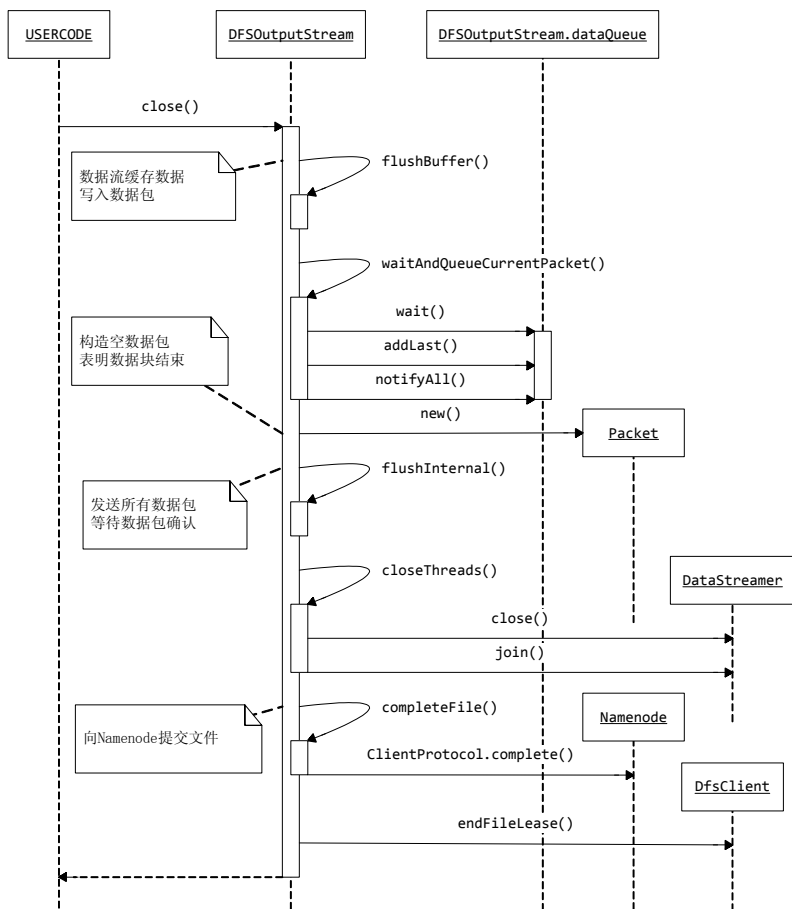


图 5-36 DFSOutputStream.close()方法调用流程图

DFSOutputStream.close()方法首先调用 `flushBuffer()` 将输出流中缓存的数据写入数据包，

然后将数据流中没有发送的数据包放入 `dataQueue` 队列中，最后构造一个新的空数据包用于标识数据块已经全部写完。

`close()`方法调用 `flushInternal()`方法确认所有数据包已经成功地写入数据流管道后，就可以调用 `closeThreads()`关闭 `DataStreamer` 线程了。之后 `close()`方法会调用 `completeFile()`方法向 `Namenode` 提交这个文件，`completeFile()`方法底层调用了 `ClientProtocol.complete()`方法。最后 `close()`方法会释放当前文件的租约。

`DFSOutputStream.close()`方法的代码如下：

```
public synchronized void close() throws IOException {
    // 状态检查
    try {
        flushBuffer();          // 将数据流中的缓存数据写入数据包
        if (currentPacket != null) {
            waitAndQueueCurrentPacket(); // 将没有发送的数据包放入 dataQueue 中
        }

        if (bytesCurBlock != 0) {
            // 构造一个空的数据包标识数据块的数据已经全部发送完毕
            currentPacket = createPacket(0, 0, bytesCurBlock, currentSeqno++);
            currentPacket.lastPacketInBlock = true;
            currentPacket.syncBlock = shouldSyncBlock;
        }

        // 确认 Datanode 已经接收了所有数据包
        flushInternal();

        ExtendedBlock lastBlock = streamer.getBlock();
        closeThreads(false); // 关闭 DataStreamer 线程
        completeFile(lastBlock); // 调用 completeFile() 向 Namenode 提交文件
        dfsClient.endFileLease(fileId); // 释放租约
    } catch (ClosedChannelException e) {
    } finally {
        closed = true;
    }
}
```

5.5 HDFS 常用工具

HDFS 提供了文件系统（File System）Shell 命令以及 `DFSAdmin` 工具来执行文件系统的常见操作，以及管理与配置 HDFS 的功能。本节我们就学习这两个常用工具的设计与实现。

5.5.1 FsShell 实现

文件系统 Shell 命令包括了许多类似于传统 Shell 的命令,这些命令会与 HDFS 进行交互,执行类似于读取文件、移动文件、创建目录等操作。在终端上执行下面的命令,可以触发文件系统 Shell 命令的执行。

```
bin/hadoop fs <args>
```

FsShell 类是 HDFS 中用于执行文件系统 Shell 命令的类,这个类的入口方法是 `main()` 方法,是一个典型的基于 `ToolRunner` 实现的应用。`FsShell.main()` 中的 `ToolRunner.run()` 方法最终会调用 `FsShell.run()` 方法, `FsShell.run()` 会调用 `CommandFactory.getInstance()` 从参数中解析出命令对应的 `Command` 对象,然后在 `Command` 对象上调用 `run()` 方法执行对应的操作。

```
public int run(String argv[]) throws Exception {
    // 初始化 FsShell
    init();

    int exitCode = -1;
    if (argv.length < 1) {
        // 打印命令如何使用
        printUsage(System.err);
    } else {
        String cmd = argv[0];
        Command instance = null;
        try {
            instance = commandFactory.getInstance(cmd);
            if (instance == null) {
                throw new UnknownCommandException();
            }
            // 调用解析出的 Command 的 run() 方法执行操作
            exitCode = instance.run(Arrays.copyOfRange(argv, 1, argv.length));
        } catch (IllegalArgumentException e) {
            displayError(cmd, e.getLocalizedMessage());
            if (instance != null) {
                printInstanceUsage(System.err, instance);
            }
        } catch (Exception e) {
            // instance.run catches IOE, so something is REALLY wrong if here
            LOG.debug("Error", e);
            displayError(cmd, "Fatal internal error");
            e.printStackTrace(System.err);
        }
    }
    return exitCode;
}
```

`Command.run()`方法会解析命令选项，扩展命令的参数，然后依次处理每个参数。
`Command.run()`方法的调用序列如下：

```
|-> processOptions(LinkedList)
\-> processRawArguments(LinkedList)
    |-> expandArguments(LinkedList)
    |    \-> expandArgument(String)
    \-> processArguments(LinkedList)
        |-> processArgument(PathData)
        |    |-> processPathArgument(PathData)
        |    \-> processPaths(PathData, PathData...)
        |        \-> processPath(PathData)
        \-> processNonexistentPath(PathData)
```

这里我们以 `Shell` 命令 `mkdir` 为例，`mkdir` 命令的实现是在 `Mkdir` 类中定义的，`Mkdir` 是 `FsCommand` 的子类。`mkdir` 命令的定义如下：

```
hdfs dfs -mkdir [-p] <paths>
```

`Mkdir` 类首先会在 `CommandFactory` 中注册 “-mkdir” 命令，用于 `FsShell` 调用 `CommandFactory.getInstance()` 方法解析出 `Mkdir` 对象。之后 `Mkdir.run()` 方法会依次调用 `processOptions()`、`processPath()` 以及 `processNonexistentPath()` 方法，`processOptions()` 会解析 `mkdir` 命令中的 -p 选项，`processPath()` 则用于处理 `mkdir` 命令中的路径参数。最后 `processNonexistentPath()` 方法则会调用 `ClientProtocol.mkdirs()` 向 `Namenode` 申请创建文件夹，完成整个文件夹的创建操作。

```
class Mkdir extends FsCommand {
    public static void registerCommands(CommandFactory factory) {
        factory.addClass(Mkdir.class, "-mkdir");
    }

    public static final String NAME = "mkdir";
    public static final String USAGE = "[-p] <path> ...";
    public static final String DESCRIPTION =
        "Create a directory in specified location.\n" +
        "-p: Do not fail if the directory already exists";

    private boolean createParents;

    @Override
    protected void processOptions(LinkedList<String> args) {
        CommandFormat cf = new CommandFormat(1, Integer.MAX_VALUE, "p");
        cf.parse(args);
        createParents = cf.getOpt("p");
    }

    @Override
    protected void processPath(PathData item) throws IOException {
        if (item.stat.isDirectory()) {
            if (!createParents) {
```

```
        throw new PathExistsException(item.toString());
    }
    } else {
        throw new PathIsNotDirectoryException(item.toString());
    }
}

@Override
protected void processNonexistentPath(PathData item) throws IOException {
    if (!item.fs.exists(new Path(item.path.toString()).getParent()) && !createParents)
    {
        throw new PathNotFoundException(item.toString());
    }
    if (!item.fs.mkdirs(item.path)) {
        throw new PathIOException(item.toString());
    }
}
```

其他的文件系统 Shell 命令的实现都类似于 `mkdir`，这里不再赘述了，请读者直接参考源码。

5.5.2 DFSAdmin 实现

HDFS 提供了 `dfsadmin` 工具，实现管理员配置与管理 HDFS 集群的功能，例如刷新 Namenode、开启快照、提交升级等操作。`dfsadmin` 命令的格式如下：

```
hdfs dfsadmin [GENERIC_OPTIONS]
```

DFSAdmin 类就是用于实现 `dfsadmin` 工具的类，它继承自 `FsShell`，也是通过 `ToolRunner.run()` 执行 `DFSAdmin.run()` 方法，不同的是 `DFSAdmin.run()` 会直接在方法体内判断命令并调用相应的处理方法。这里我们以 `allowSnapshot` 命令为例，这个命令用于在 HDFS 指定目录下开启快照功能。`allowSnapshot` 命令的定义如下：

```
hdfs dfsadmin-allowSnapshot <snapshotDir>
```

`DFSAdmin.run()` 方法会调用 `allowSnapshot()` 方法，`allowSnapshot()` 会调用 `DistributedFileSystem.allowSnapshot()` 执行开启快照的操作。`allowSnapshot()` 方法的实现代码如下：

```
public void allowSnapshot(String[] argv) throws IOException {
    DistributedFileSystem dfs = getDFS();
    try {
        dfs.allowSnapshot(new Path(argv[1]));
    } catch (SnapshotException e) {
        throw new RemoteException(e.getClass().getName(), e.getMessage());
    }
    System.out.println("Allowing snapshot on " + argv[1] + " succeeded");
}
```

DFSAdmin 的其他方法的实现类似于 `allowSnapshot()` 方法，请读者直接参考源码。

参考文献

参考书籍

- [1] 蔡斌, 陈湘萍. Hadoop 技术内幕: 深入解析 Hadoop Common 和 HDFS 架构设计与实现原理[M]. 北京: 机械工业出版社, 2013
- [2] 周敏奇, 王晓玲, 金澈清, 译. Tom White. Hadoop 权威指南[M]. 2 版. 北京: 清华大学出版社, 2011
- [3] Eric Freeman, Bert Bates. 深入浅出设计模式[M]. 南京: 东南大学出版社, 2005
- [4] Bruce Eckel. Java 编程思想[M]. 北京: 机械工业出版社, 2007

参考 HDFS JIRA

- [1] HDFS-265: Revisit append
- [2] HDFS-1052: Hdfs scalability with multiple namenodes
- [3] HDFS-1073: Simpler model for Namenode'sFsImage and EditLogs
- [4] HDFS-1580: Add interface for generic Write Ahead Logging mechanism
- [5] HDFS-1623: High Availability Framework for HDFS NN
- [6] HDFS-2185: HA: HDFS portion of ZK-based FailoverController
- [7] HDFS-3042: Automatic failover support for NN HA
- [8] HDFS-3077: Quorum-based protocol for reading and writing edit logs

参考网络资源

- [1] Protocol Buffer 官方主页: <https://github.com/google/protobuf>
- [2] Hadoop 2.6 官方文档: <http://hadoop.apache.org/docs/r2.6.0/>
- [3] hortonworks 官方博客: <http://zh.hortonworks.com/blog/>
- [4] cloudera 官方博客: <http://blog.cloudera.com/blog/>

- [5] Java NIO 入门: <http://www.ibm.com/developerworks/cn/education/java/j-nio/j-nio.html>
- [6] Scalable IO in Java: <http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf>
- [7] caibin 博客: <http://caibin.iteye.com>
- [8] datascientis 博客: <http://yanbohappysinaapp.com>
- [9] dongxicheng 博客: <http://dongxicheng.org>
- [10] 其他引用
 - <http://www.iteblog.com>
 - <http://bearsorry.iteye.com>
 - <http://www.cnblogs.com/xia520pi>
 - <http://www.cnblogs.com/foxmailed>
 - <http://blog.csdn.net/anzhsoft>
 - <http://blog.csdn.net/fcbayernmunchen/>
 - <http://blog.csdn.net/danssion>
 - <http://blog.csdn.net/zhangjun2915>
 - <http://blog.csdn.net/liuhong1123>
 - <http://blog.csdn.net/chenpingbupt/>
 - <http://www.wuzesheng.com>
 - <http://zengzhaozheng.blog.51cto.com>